Decidability of downward XPath

DIEGO FIGUEIRA
INRIA, ENS Cachan, LSV & University of Edinburgh

We investigate the satisfiability problem for downward-XPath, the fragment of XPath that includes the child and descendant axes, and tests for (in)equality of attributes' values. We prove that this problem is decidable, ExpTime-complete. These bounds also hold when path expressions allow closure under the Kleene star operator. To obtain these results, we introduce a Downward Data automata model (DD automata) over trees with data, which has a decidable emptiness problem. Satisfiability of downward-XPath can be reduced to the emptiness problem of DD automata and hence its decidability follows. Although downward-XPath does not include any horizontal axis, DD automata are more expressive and can perform some horizontal tests. Thus, we show that the satisfiability remains in ExpTime even in the presence of the regular constraints expressible by DD automata. However, the same problem in the presence of any regular constraint is known to have a non-primitive recursive complexity. Finally, we give the exact complexity of the satisfiability problem for several fragments of downward-XPath.

Categories and Subject Descriptors: I.7.2 [**Document Preparation**]: Markup Languages; H.2.3 [**Database Management**]: Languages; H.2.3 [**Languages**]: Query Languages

General Terms: Algorithms, Languages

Additional Key Words and Phrases: data tree, XML, XPath, downward data automata, DD automata, data values, infinite alphabet, emptiness, satisfiability, extensible language

1. INTRODUCTION

XPath is arguably the most widely used XML query language. It is implemented in XSLT and XQuery and it is used as a constituent part of several specification and update languages. XPath is fundamentally a general purpose language for addressing, searching, and matching pieces of an XML document. It is an open standard and constitutes a World Wide Web Consortium (W3C) Recommendation [Clark and DeRose 1999].

Query containment and query equivalence are important static analysis problems, which are useful to query optimization tasks. In logics closed under boolean operators, these problems reduce to checking for *satisfiability*: Is there a document on which a given query has a non-empty result? By answering this question we can decide at compile time whether the query contains a contradiction and thus the computation of the query (or subquery) on the document can be avoided. Or,

Author's address: 10 Crichton Street, Edinburgh EH8 9AB, UK.
The author acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under the FET-Open grant agreement FOX, number FP7-ICT-233599.

by answering the query equivalence problem, one can test if a query can be safely replaced by another one which is more optimized in some sense (*e.g.*, in the use of some resource). Moreover, the satisfiability problem is crucial for applications on security [Fan et al. 2004], type checking transformations [Martens and Neven 2007], and consistency of XML specifications.

Core-XPath (term introduced in [Gottlob et al. 2005]) is the fragment of XPath 1.0 that captures all the navigational behavior of XPath. It has been well studied and its satisfiability problem is known to be decidable even in the presence of DTDs. The extension of this language with the possibility to make equality and inequality tests between attributes of elements in the XML document is named Core-Data-XPath in [Bojańczyk et al. 2009]. The satisfiability problem for this logic is undecidable, as shown in [Geerts and Fan 2005]. It is then reasonable to study the interaction between different navigational fragments of XPath with equality tests with the aim of finding decidable and computationally well-behaved fragments. In the present work, we focus on the downward-looking fragments of XPath, from now on denoted by $\mathsf{XPath}(\downarrow_*, \downarrow, =)$, where navigation between nodes can only be done in the downward direction: from the root towards the leaves. With this logic one can express, for example, that every node labeled a has a descendant b with the same data value, or that there are no two descendant nodes labled with c and d with the same data value. We remark that the logics we treat here correspond to subsets of the W3C standard XPath 1.0 [Clark and DeRose 1999], for a complete formal semantics see [Gottlob et al. 2002]. XPath 2.0 is a radically different language than XPath 1.0. It integrates XPath and XQuery in a common syntax, and includes variables and explicit quantification. These constructs are out of the scope of the current work.

Our main contribution is that the satisfiability problem for $\mathsf{XPath}(\downarrow_*, \downarrow, =)$ is decidable. This is the fragment with equality and inequality tests of attributes' values, the $\downarrow_*$ axis that can access descendant nodes at any depth and the $\downarrow$ axis to access child elements.[1] Actually, we prove a stronger result, showing the decidability of the satisfiability of $\mathsf{regXPath}(\downarrow, =)$, which is the extension of $\mathsf{XPath}(\downarrow_*, \downarrow, =)$ with the Kleene star operator to take reflexive-transitive closures of arbitrary path expressions. We nail down the precise complexity showing an ExpTime decision procedure. The problem is hence ExpTime-complete since $\mathsf{XPath}(\downarrow, \downarrow_*)$ is already ExpTime-hard by [Benedikt et al. 2008]. To obtain this bound, we introduce the class of Downward Data automata (DD automata). We show that any $\mathsf{regXPath}(\downarrow, =)$ formula can be efficiently translated to an equivalent DD automaton. This automata model has a 2ExpTime emptiness problem, but can be shown to be decidable in ExpTime when restricted to the sub-class of automata needed to capture $\mathsf{regXPath}(\downarrow, =)$. In this way we obtain an ExpTime procedure for the satisfiability of $\mathsf{regXPath}(\downarrow, =)$.

In fact, DD automata are more expressive than $\mathsf{regXPath}(\downarrow, =)$, and this is true even for the aforementioned sub-class of automata. For example, although $\mathsf{regXPath}(\downarrow, =)$ does not include any horizontal axis, DD automata can test for

[1]We use the notation $\mathsf{XPath}(\mathcal{O}, =)$ to refer to the fragment of XPath including all the relations contained in $\mathcal{O}$ for navigation, and where the presence of '=' indicates that the language can perform data tests (both for equality and inequality of data values).

certain horizontal properties. This model can express, for instance, that the sequence of children of the root is described by the regular expression $(a\,b\,c)^*$. It will then follow that the satisfiability problem for regXPath($\downarrow, =$) under the regular constraints that can be expressed by DD automata remains decidable in ExpTime. This is a particularly well behaved class of regular properties, since the satisfiability problem of regXPath($\downarrow, =$) restricted to regular tree languages is known to have non-primitive recursive complexity.[2]

On the other hand, we prove that the fragment XPath($\downarrow_*, =$) without the $\downarrow$ axis is ExpTime-hard, even for a restricted fragment of XPath($\downarrow_*, =$) without unions of path expressions. This reduction seems to rely on data equality tests, as the corresponding fragment XPath($\downarrow_*$) without unions is shown to be PSpace-complete. We thus prove that the satisfiability problems for XPath($\downarrow_*, =$), XPath($\downarrow_*, \downarrow, =$) and regXPath($\downarrow, =$) are all ExpTime-complete. Additionally, we present a natural fragment of XPath($\downarrow_*, =$) that is PSpace-complete. We complete the picture by showing that satisfiability for XPath($\downarrow, =$) is also PSpace-complete. Our results, together with the results of [Benedikt et al. 2008; Marx 2004], establish the precise complexity for all downward fragments of XPath with and without data tests (*cf.* Table IV on page 44).

Related work

The main results of this work first appeared in the conference paper [Figueira 2009]. The cited work does not contain a full proof of the decidability of the downward fragment of XPath, but only the main ideas due to a space limitation. Here we give a full and detailed proof by a reduction to a powerful class of automata, and we extend some results. Although the main XPath results of [Figueira 2009] are the same as of the present work, the underlying automata model is completely different. There are basically two reasons to adopt a different strategy to show the decidability. First, the automaton introduced here is simpler than the one of [Figueira 2009]: it does not require a nested definition between two different kinds of automata as in [Figueira 2009]. And second, it is more general: it can express data properties that cannot be expressed in the model of [Figueira 2009], and it can test (weak) regular properties on the sequence of children of a node. The larger expressive power of the automata model yields a decidability procedure for the satisfiability of downward-XPath under a subclass of regular properties, something that was out of the scope of [Figueira 2009]. The results of this paper also appear in the doctoral dissertation [Figueira 2010b, Chapter 5].

Benedikt et al. [2008] study the satisfiability problem for many XPath logics, mostly fragments without negation or without data equality tests. Also, the fragment XPath($\downarrow, =$) is shown to be in NExpTime. We improve this result by providing an optimal PSpace upper bound. It is also known that XPath($\downarrow$) is already PSpace-hard[3], and in this work we give a matching upper bound showing PSpace-completeness. Furthermore, Marx [2004] proves that XPath($\downarrow, \downarrow_*$) is

[2]This is a corollary of [Figueira and Segoufin 2009].
[3]This is a direct consequence of XPath($\downarrow$) being able to code any formula from the normal modal logic K, which enjoys the finite- and tree-model properties [Blackburn et al. 2001], and is PSpace-complete [Ladner 1977].

EXPTIME-complete. In this work we prove that this complexity is preserved in the presence of data values and even under closure with Kleene star. We also study $\mathsf{XPath}(\downarrow_*, =)$, a fragment that is not considered in [Benedikt et al. 2008], and show that $\mathsf{XPath}(\downarrow_*, =)$ is EXPTIME-complete while $\mathsf{XPath}(\downarrow_*)$ is PSPACE-complete. In this case, data tests do make a real difference in complexity (at least under widely-held complexity theoretical assumptions).

First-order logic with two variables and data equality tests is investigated in [Bojańczyk et al. 2009]. Although in the absence of data values FO^2 is expressive-equivalent to Core-XPath (*cf.* [Marx 2005]), FO^2 with data equality tests is incomparable with respect to all the data aware fragments treated here. Bojańczyk et al. [2009] also showed the decidability of a fragment of $\mathsf{XPath}(\uparrow, \downarrow, \leftarrow, \rightarrow, =)$ with sibling and upward axes but restricted to local elements accessible by a 'one-step' relation, *i.e.*, parent, child, previous-sibling, next-sibling. This fragment is restricted to using data formulæ of the kind $\langle \varepsilon = \alpha \rangle$ (or $\langle \varepsilon \neq \alpha \rangle$) that test whether a data value accessible with a path expression α is equal to the data value of the *current* point of evaluation (denoted by ε). In contrast, the fragments we treat here disallow upward and sibling axes but allow the descendant $\downarrow_*$ axis and arbitrary $\langle \alpha = \alpha' \rangle$, $\langle \alpha \neq \alpha' \rangle$ data test expressions.

In [Figueira 2010a] the fragment $\mathsf{XPath}(\downarrow, \downarrow_*, \rightarrow, \rightarrow^*, =)$ is treated, denominated 'forward-XPath'. In the cited work, the full set of downward (child, descendant) and rightward (next-sibling, following-sibling) axes are allowed. This logic can express, for example, a *key constraint* property on a label a (*i.e.*, that all the nodes labeled with a have different data values), which cannot be expressed in downward-XPath. The satisfiability problem for forward-XPath is shown to still be decidable in [Figueira 2010a]. Also, in [Figueira and Segoufin 2011], the satisfiability problem for $\mathsf{XPath}(\downarrow, \downarrow_*, \uparrow, \uparrow^*, =)$ is shown to be decidable. This fragment is an extension of the downward fragment with the parent ($\uparrow$) and ancestor ($\uparrow^*$) relations, and it is called *vertical*-XPath. However, these decidability results come at the expense of a huge rise in complexity with respect to the downward fragment. Indeed, the time or space required by the decidability procedure for the satisfiability of the forward or vertical fragments cannot be bounded by any primitive recursive function [Jurdziński and Lazić 2008]. In fact, this non-primitive recursive lower-bound holds for any XPath fragment containing (or being able to express) any of the axes $\rightarrow^+$, $\uparrow^+$, ${}^+\!\leftarrow$ as shown in [Figueira and Segoufin 2009].[4] However, all the downward fragments are below EXPTIME, as evidenced in our present work. Meeting these elementary upper bounds requires an altogether different approach from the ones taken in [Figueira 2010a; Figueira and Segoufin 2011].

2. PRELIMINARIES

Notation

We first fix some basic notation. Let $\mathbb{N} = \{1, 2, 3, \dots\}$, and let $[n] := \{1, \dots, n\}$ for $n \in \mathbb{N}$. By $\wp(S)$ we denote the power set of a set S. We use letters $\mathbb{A}$, $\mathbb{B}$ to denote finite alphabets, $\mathbb{D}$ to denote a countably infinite domain (*e.g.*, $\mathbb{N}$) and the letters $\mathbb{E}$ and $\mathbb{F}$ to denote any kind (finite or infinite) of set. In our examples we

[4]Here, $(\cdot)^+$ stands for the non-reflexive version of $(\cdot)^*$.

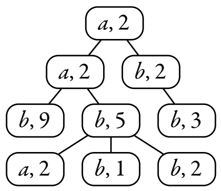

Fig. 1. A data tree of $\mathit{Trees}(\mathbb{A}\times\mathbb{D})$ with $\mathbb{A} = \{a, b\}$ and $\mathbb{D} = \mathbb{N}$.

will consider $\mathbb{D} = \mathbb{N}$. We define the composition $(f \circ g)(x) = f(g(x))$.

Unranked ordered finite trees

We define $\mathit{Trees}(\mathbb{E})$, the set of finite, ordered and unranked trees over an arbitrary alphabet $\mathbb{E}$.

A **position** of a tree is an element of $\mathbb{N}^*$. The root's position is the empty string and we denote it by ϵ. The position of any other node in the tree is the position of its parent appended to a number $s + 1 \in \mathbb{N}$, where s is the number of the node's left siblings. We write '$\cdot$' for the concatenation operator, and we use x, y, z, w, v as variables for positions, and i, j, k, l, m, n as variables for numbers. Thus, for example, $x{\cdot}i$ is a position which is not the root, that has $i - 1$ siblings to the left, and has x as parent position.

Formally, we define $\mathsf{TreesPos} \subseteq \wp(\mathbb{N}^*)$ the set of sets of finite tree positions, such that: $P \in \mathsf{TreesPos}$ iff (a) $P \subseteq \mathbb{N}^*, |P| < \infty$; (b) it is prefix-closed; and (c) if $n{\cdot}(i + 1) \in P$ for $i \in \mathbb{N}$, then $n{\cdot}i \in P$. A tree is a mapping from a set of positions to letters of the alphabet

$$\mathit{Trees}(\mathbb{E}) := \{\mathbf{t} : P \to \mathbb{E} \mid P \in \mathsf{TreesPos}\} .$$

Given a tree $\mathbf{t} \in \mathit{Trees}(\mathbb{E})$, $\mathsf{pos}(\mathbf{t})$ denotes the domain P and $\mathsf{alph}(\mathbf{t})$ the alphabet $\mathbb{E}$ of the tree. From now on, we informally refer by 'node' to a position x together with the value $\mathbf{t}(x)$. We define the *ancestor* partial order $\preceq$ as the prefix relation $x \preceq x{\cdot}y$ for every x, y, and $\prec$ as the strict prefix relation $x \prec x{\cdot}y$ for $|y| > 0$. We say that two positions x, y are **incomparable** if $x \not\preceq y$ and $y \not\preceq x$. Given a tree $\mathbf{t}$ and $x \in \mathsf{pos}(\mathbf{t})$, $\mathbf{t}|_x$ denotes the subtree of $\mathbf{t}$ at position x. That is, $\mathbf{t}|_x : \{y \mid x{\cdot}y \in \mathsf{pos}(\mathbf{t})\} \to \mathsf{alph}(\mathbf{t})$ is the tree $\mathbf{t}|_x(y) = \mathbf{t}(x{\cdot}y)$. The notation $f|_x$ is generalized to any function f whose domain is in $\mathsf{TreesPos}$. In the context of a tree $\mathbf{t}$, a **siblinghood** is a maximal sequence of siblings or, in other words, a sequence of positions $x{\cdot}1, \ldots, x{\cdot}l \in \mathsf{pos}(\mathbf{t})$ such that $x{\cdot}(l + 1) \notin \mathsf{pos}(\mathbf{t})$. Given a tree $\mathbf{t}$ and a position x, we define $\#\mathit{children}(\mathbf{t}, x) = |\{x{\cdot}i \mid x{\cdot}i \in \mathsf{pos}(\mathbf{t})\}|$.

Given two trees $\mathbf{t}_1 \in \mathit{Trees}(\mathbb{E})$, $\mathbf{t}_2 \in \mathit{Trees}(\mathbb{F})$ with the same set of positions P (*i.e.*, $\mathsf{pos}(\mathbf{t}_1) = \mathsf{pos}(\mathbf{t}_2) = P$), we define $\mathbf{t}_1 \otimes \mathbf{t}_2 : P \to (\mathbb{E}\times\mathbb{F})$ by $(\mathbf{t}_1 \otimes \mathbf{t}_2)(x) = (\mathbf{t}_1(x), \mathbf{t}_2(x))$.

The set of **data trees** over a finite alphabet $\mathbb{A}$ and an infinite domain $\mathbb{D}$ is defined as $\mathit{Trees}(\mathbb{A}\times\mathbb{D})$. Note that every tree $\mathbf{t} \in \mathit{Trees}(\mathbb{A}\times\mathbb{D})$ can be decomposed into two trees $\mathbf{a} \in \mathit{Trees}(\mathbb{A})$ and $\mathbf{d} \in \mathit{Trees}(\mathbb{D})$ such that $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$. Figure 1 shows an example of a data tree. We denote the set of data values used in a data tree with

$$\mathit{data}(\mathbf{a} \otimes \mathbf{d}) := \{\mathbf{d}(x) \mid x \in \mathsf{pos}(\mathbf{d})\} .$$

We freely extend this notation and denote by $data(X)$ all the elements of $\mathbb{D}$ contained in X, for whatever object X may be.

Regular properties on branches

One important object of a data tree is that of a 'branch': a succession of nodes that starts at the root and goes downward, ending at any node of the tree. Note that we consider the possibility that a branch may end at an inner node and not necessarily at a leaf. Given two positions $x \prec y$ in a branch, we define $\mathrm{str}(x, y)$ as the string of labels from the finite alphabet contained between x and y, including x and y. All the power of the automata model we will define relies on the ability to test data properties at distant positions of the tree. These positions are end points of branches whose string belongs to a certain regular language. We next define the execution of a nondeterministic finite automaton over a branch of a data tree.

NFAs $\mathcal{A} = (\mathbb{A}, Q, q_1, \delta, F)$ are defined as usual, that is, $\mathbb{A}$ is a finite alphabet, Q is a finite set of states, q_1 is the initial state, $\delta \subseteq Q \times \mathbb{A} \times Q$ is the transition function, and $F \subseteq Q$ is the final set of states. As usual, $L(\mathcal{A})$ denotes the set of strings accepted by $\mathcal{A}$. We write $(q, a, q') \in \mathcal{A}$ to denote that there is a transition from q to q' in $\mathcal{A}$ by reading a.

Let $\mathbf{t} = \mathbf{a} \otimes \mathbf{b}$ be a data tree in $\mathit{Trees}(\mathbb{A} \times \mathbb{D})$. Given a NFA $\mathcal{A}$ over the alphabet $\mathbb{A}$, we consider the execution of $\mathcal{A}$ on the string contained between a position $x \in \mathsf{pos}(\mathbf{t})$ and a descendant position $x \cdot y \in \mathsf{pos}(\mathbf{t})$. For a state q, let $\mathcal{A}[q]$ be identical to $\mathcal{A}$ with the exception that now q is the initial state.

We note a 'one-step' of the execution as

$$q \xrightarrow[x]{\mathcal{A}} q'$$

if $(q, \mathbf{a}(x), q')$ is a transition of $\mathcal{A}$, the data tree being implicit in the notation. And we write

$$q \xrightarrow[x, x\cdot y]{\mathcal{A}} q' \quad \text{iff} \quad \begin{cases} q \xrightarrow[x]{\mathcal{A}} q' & \text{if } y = \epsilon; \\ q \xrightarrow[x]{\mathcal{A}} p_1 \xrightarrow[x\cdot i_1]{\mathcal{A}} p_2 \xrightarrow[x\cdot i_1 \cdot i_2]{\mathcal{A}} \cdots \xrightarrow[x\cdot i_1 \cdots i_n]{\mathcal{A}} p_{n+1} = q' & \text{if } y = i_1 \cdots i_n. \end{cases}$$

That is, if we can reach the configuration by reading the letters between x and $x \cdot y$ in a descending way. Note that the automaton's run includes the starting and ending labels. Hence, all runs execute at least one transition.

We fix a notation for the data values of those positions selected by some run of $\mathcal{A}$. For a state q of $\mathcal{A}$ and a tree $\mathbf{t}$, we write $[\![\mathcal{A}, q]\!]^{\mathbf{t}}$ to denote the set of data values of all positions x that can be reached starting at the root with state q and ending at x with a final state from F,

$$[\![\mathcal{A}, q]\!]^{\mathbf{t}} = \{\mathbf{d}(x) \mid x \in \mathsf{pos}(\mathbf{t}), \mathrm{str}(\epsilon, x) \in L(\mathcal{A}[q])\} \ .$$

We write $[\![\mathcal{A}]\!]^{\mathbf{t}}$ for $[\![\mathcal{A}, q_1]\!]^{\mathbf{t}}$, where q_1 is the initial state of $\mathcal{A}$.

3. AUTOMATA MODEL

In this section we define an automata model that runs over data trees. We show that this model has a decidable emptiness problem in Section 4. In Section 6.1 we show that $\mathsf{XPath}(\downarrow_*, \downarrow, =)$ formulæ can be effectively translated to an equivalent DD automaton, thus obtaining a decidability procedure for its satisfiability problem.

Our model, called *Downward Data automaton* (or simply *DD automaton*), has an execution that consists of two steps: (1) the execution of a transducer, and (2) the verification of data properties of the transduced tree.

For a data tree $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$, the first step consists in the translation of $\mathbf{a}$ into another tree $\mathbf{b}$. This is done using a nondeterministic letter-to-letter transducer over unranked trees. We adopt a more detailed definition, where the transducer explicitly has as a parameter the class $\mathscr{C}$ of regular properties that it can test over a siblinghood at each transition. If we take this parameter to be the set of all regular properties, this automaton is a standard transducer over unranked trees. However, the emptiness problem for Downward Data automata has a very high complexity lower bound unless we restrict $\mathscr{C}$ to be a suitable subclass of regular languages (*cf.* discussion on page 9). This class, defined as the set of *extensible* languages, will be introduced in the sequel (Definition 3.7).

In the second step, for every subtree of the transduced tree $\mathbf{b} \otimes \mathbf{d}$, a property on the data values of the tree is verified. The letter at the root of the subtree under inspection determines the property to verify. The properties are boolean combinations of tests verifying the existence of data values shared by nodes in the subtree, hanging from branches satisfying some regular expression.

A brief comment on notation. In the definition of these automata there will be two sorts of sets of states, namely the states corresponding to the run of the transducer, and the states corresponding to the run of the verifier. To avoid confusion we consistently write $\dot{Q}, \dot{q}, \dot{q}', \dots$ as symbols for the states of the transducer, and $Q, q, q', \dots$ for the states of the verifier.

Transducer

Let $\mathscr{C}$ be a subclass of regular languages $\mathscr{C} \subseteq \mathit{REG}$. We define a bottom-up unranked tree transducer. This definition is parametrized by $\mathscr{C}$ in the following sense: At every transition, the automaton can test the siblinghood for membership in some regular language that must be in the class $\mathscr{C}$.

Definition 3.1. A **$\mathscr{C}$-transducer** defines a relation between trees with the same set of positions. Given a transducer $\mathcal{R}$, we use the same symbol $\mathcal{R}$ to denote the relation of the pairs of trees accepted by $\mathcal{R}$ with a slight abuse of notation, *i.e.*, $\mathcal{R} \subseteq \mathit{Trees}(\mathbb{A} \times \mathbb{B})$. The idea is that $\mathbf{a} \in \mathit{Trees}(\mathbb{A})$ and $\mathbf{b} \in \mathit{Trees}(\mathbb{B})$ are related by $\mathcal{R}$ if $\mathbf{a} \otimes \mathbf{b} \in \mathcal{R}$. Thus, in order to be related by $\mathcal{R}$, the trees $\mathbf{a}$ and $\mathbf{b}$ need to have the same set of positions. This relation is defined as the set of accepted trees of a nondeterministic bottom-up unranked transducer represented as a tuple $\langle \mathscr{C}, \mathbb{A}, \mathbb{B}, \dot{Q}, \dot{Q}_F, \delta \rangle$ where

—$\mathbb{A}$ and $\mathbb{B}$ are finite alphabets of letters,

—$\dot{Q}$ is a finite set of states,

—$\dot{Q}_F \subseteq \dot{Q}$ is the set of final states,

—$\mathscr{C} \subseteq \mathit{REG}(\dot{Q})$ is a class of regular languages over the alphabet $\dot{Q}$,

—$\delta \subseteq \dot{Q} \times \mathbb{A} \times \mathbb{B} \times \mathscr{C}$ is a finite set of transitions.

We define $\mathbf{a} \otimes \mathbf{b} \in \mathcal{R}$ iff $\mathbf{a} \in \mathit{Trees}(\mathbb{A})$, $\mathbf{b} \in \mathit{Trees}(\mathbb{B})$ with $\mathsf{pos}(\mathbf{a}) = \mathsf{pos}(\mathbf{b}) = P$, and there exists a state assigment $\rho : P \to \dot{Q}$, such that

—for every leaf x, there exists $\mathcal{L} \in \mathscr{C}$ with $\epsilon \in \mathcal{L}$ and $(\rho(x), \mathbf{a}(x), \mathbf{b}(x), \mathcal{L}) \in \delta$,

—for every siblinghood $x{\cdot}1, \ldots, x{\cdot}l$, there is $\mathcal{L} \in \mathscr{C}$ such that $(\rho(x), \mathbf{a}(x), \mathbf{b}(x), \mathcal{L}) \in \delta$ and $\rho(x{\cdot}1) \cdot \cdots \cdot \rho(x{\cdot}l) \in \mathcal{L}$.

We call ρ a **run** of $\mathcal{R}$ on $\mathbf{a} \otimes \mathbf{b}$. We say that the run is **accepting** iff $\rho(\epsilon) \in \dot{Q}_F$.

We now turn to the second step.

Verifier

Definition 3.2. A **verifier** $\mathcal{V} \subseteq \mathit{Trees}(\mathbb{B}{\times}\mathbb{D})$ defines a set of data trees that are valid with respect to some data properties. It is a tuple $\langle \mathcal{A}_1, \ldots, \mathcal{A}_{\mathsf{K}}, \mathsf{v} \rangle$ of K NFA over the alphabet $\mathbb{B}$, namely $\mathcal{A}_1, \ldots, \mathcal{A}_{\mathsf{K}}$, and a function $\mathsf{v} : \mathbb{B} \to \Phi$ mapping letters of the alphabet to formulæ expressing data properties. The idea is that every subtree $\mathbf{t}|_x$ of the original tree $\mathbf{t}$ must verify a property expressed by the formula $\mathsf{v}(\mathbf{b}(x))$. A typical property that we can test at a subtree is the existence of two positions with the same data value, such that one is reachable by going downward through a branch whose labelling is in some regular language $\mathcal{L}_1$, and the other by some other branch with labelling in $\mathcal{L}_2$.

The properties of Φ are a subset of closed first-order formulæ with no quantifier alternation (that is, no $\forall\exists$ or $\exists\forall$ patterns allowed), and K unary relations, one for every automaton $\mathcal{A}_i$, namely

$$D_1, \ldots, D_{\mathsf{K}} \ .$$

Given a set *Vars* of variables, Φ contains all the formulæ ϕ defined by the grammar

$$\begin{aligned} \phi &::= \neg\,\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \exists \bar{v}.\psi \\ \psi &::= \psi \wedge \psi \mid \psi \vee \psi \mid v = v' \mid v \neq v' \mid D_i(v) \end{aligned}$$

where $\bar{v}$ stands for a set of variables from *Vars*, $v, v' \in \mathit{Vars}$, $i \in [\mathsf{K}]$, and we restrict ϕ to have no free variables. The variables are interpreted over the set of data values, and the D_i's as sets of values reachable by the automata $\mathcal{A}_i$'s.

Given a data tree $\mathbf{t}$, let $\mathcal{I}_{\mathbf{t}}$ be the first-order interpretation where each unary relation D_i is interpreted as $[\![\mathcal{A}_i]\!]^{\mathbf{t}}$, and the interpretation of the domain is $\mathbb{D}$. We say that that φ is verified in $\mathbf{t}$ if φ is true under the interpretation $\mathcal{I}_{\mathbf{t}}$. A verifier $\langle \mathcal{A}_1, \ldots, \mathcal{A}_{\mathsf{K}}, \mathsf{v} \rangle$ *accepts* a data tree $\mathbf{t} = \mathbf{a} \otimes \mathbf{d} \in \mathit{Trees}(\mathbb{B}{\times}\mathbb{D})$ if for every position $x \in \mathsf{pos}(\mathbf{t})$ the formula $\mathsf{v}(\mathbf{a}(x))$ is verified in the subtree $\mathbf{t}|_x$.

Example 3.3. Suppose $\mathsf{v}(a) = \exists\, x_1, x_2 \ . \ D_1(x_1) \wedge D_2(x_2) \wedge x_1 \neq x_2$, for a letter a. This means that for every a-rooted subtree we can find two branches in the regular languages recognized by $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively leading to two nodes whose data values are different as depicted in Figure 2.

Definition 3.4. A $\mathscr{C}$-**Downward Data automaton** ($\mathscr{C}$-DD for short) is a pair $(\mathcal{R}, \mathcal{V})$ made of a $\mathscr{C}$-transducer $\mathcal{R} \subseteq \mathit{Trees}(\mathbb{A}{\times}\mathbb{B})$ and a verifier $\mathcal{V} \subseteq \mathit{Trees}(\mathbb{B}{\times}\mathbb{D})$. A data tree $\mathbf{a} \otimes \mathbf{d}$ is accepted by $(\mathcal{R}, \mathcal{V})$ iff there exists $\mathbf{b} \in \mathit{Trees}(\mathbb{B})$ such that $\mathbf{a} \otimes \mathbf{b} \in \mathcal{R}$ and $\mathbf{b} \otimes \mathbf{d} \in \mathcal{V}$, as depicted in Figure 3. When we want to make explicit that the witnessing tree for the acceptance of $\mathbf{a} \otimes \mathbf{d}$ is $\mathbf{b}$, we will say equivalently

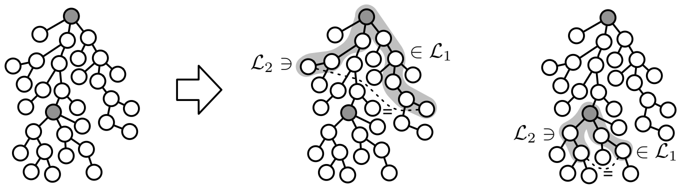

Fig. 2. If $\mathcal{L}_1, \mathcal{L}_2$ are the languages recognized by $\mathcal{A}_1, \mathcal{A}_2$, and the marked positions are labeled by a, then $\mathsf{v}(a)$ is verified in both subtrees.

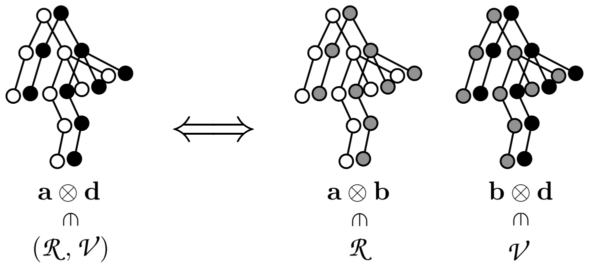
$\mathbf{a} \otimes \mathbf{d}$
$\Cap$
$(\mathcal{R}, \mathcal{V})$
$\Longleftrightarrow$
$\mathbf{a} \otimes \mathbf{b}$
$\Cap$
$\mathcal{R}$
$\mathbf{b} \otimes \mathbf{d}$
$\Cap$
$\mathcal{V}$

Fig. 3. Acceptance condition of a DD automaton $(\mathcal{R}, \mathcal{V})$.

that $\mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$ is accepted by $(\mathcal{R}, \mathcal{V})$. Also, we say that $\mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$ *has a run* if there is a run of $\mathcal{R}$ on $\mathbf{a} \otimes \mathbf{b}$, where $\mathbf{b} \otimes \mathbf{d} \in \mathcal{V}$.

We now give some closure properties of DD automata.

We say that a class of languages $\mathscr{C}$ is closed under **componentwise product**, if for every two languages $\mathcal{L}_1, \mathcal{L}_2$ of $\mathscr{C}$ and the language $\mathcal{L}_1 \times_c \mathcal{L}_2 = \{(a_1, b_1) \cdots (a_n, b_n) \mid a_1 \cdots a_n \in \mathcal{L}_1, b_1 \cdots b_n \in \mathcal{L}_2\}$ is in $\mathscr{C}$. Note that if $\mathcal{L}_1, \mathcal{L}_2$ are regular, $\mathcal{L}_1 \times_c \mathcal{L}_2$ is regular.

PROPOSITION 3.5. *Given a class $\mathscr{C}$ of languages closed under componentwise product, the class of languages recognized by $\mathscr{C}$-DD automata is closed under intersection.*

The proof of this proposition can be found in the online appendix.

To obtain closure under complementation we need some extra hypothesis. Given an alphabet $\mathbb{A}$ and a letter $a \in \mathbb{A}$, we call the **membership language of** a to the language $\{w \cdot a \cdot w' \mid w, w' \in \mathbb{A}^*\}$. We also say that a class $\mathscr{C} \subseteq REG$ is **closed under inverse homomorphisms** if for every language $\mathcal{L} \in \mathscr{C}$ over an alphabet $\mathbb{A}$ and for every homomorphism $h : \mathbb{B} \to \mathbb{A}$ there is a language $\mathcal{L}' \in \mathscr{C}$ over $\mathbb{B}$ such that $\mathcal{L}' = \{w \in \mathbb{B}^* \mid h(w) \in \mathcal{L}\}$.

PROPOSITION 3.6. *Let $\mathscr{C}$ a class of languages closed under intersection, complementation, inverse homomorphisms and containing all membership languages. The class of languages recognized by $\mathscr{C}$-DD automata is closed under complementation.*

The proof of this proposition can be found in the online appendix.

Complexity of emptiness problem. The emptiness problem for $\mathscr{C}$-Downward Data automata has a non-primitive recursive complexity if we do not impose any restriction to $\mathscr{C}$. The non-primitive recursive lower bound can be seen as a consequence of the fact that DD automata

—can force the model to be linear (*i.e.*, that all nodes have at most one child), and
—can capture any downward XPath formula, as we will see in Section 6.1.

It is known that the satisfiability problem for downward XPath on non-branching data trees (called *data word*) is non-primitive recursive (see *e.g.* [Figueira and Segoufin 2009]). Although the emptiness problem for DD automata without any restriction is not discussed here, we believe that it is decidable. Probably, it can be shown to be decidable via the theory of well quasi orderings, by a similar technique as the one used in [Figueira 2010a] in the context of downward alternating register automata.

Nevertheless, when $\mathscr{C}$ is restricted to have some good properties, emptiness of DD automata can be tested in 2ExpTime. Further, if the verifier is such that the number of occurrences of the relations D_i inside every quantified subformula is bounded by a constant[5], we achieve an ExpTime decision procedure. We will show that downward XPath formulæ can be translated in PTime into equivalent DD automata. These automata are restricted to a class of languages $\mathscr{C}$ with good properties, and such that the number of occurrences of the D_i are always bounded by 2. Then, the decidability in ExpTime of the satisfiability problem for downward-XPath will follow. In the next section we define which is the necessary property that $\mathscr{C}$ must have in order to obtain the aforementioned upper bounds.

Extensibility of languages

To have a low complexity in the testing for emptiness of $\mathscr{C}$-DD automata, we need to weaken the kind of regular properties that the transducer can verify. That is, we need to restrict $\mathscr{C}$. The idea is that if a word is in a language, then an *extension* of the word where more occurrences of each letter may occur must also be in the language. Let us formally define this notion.

Let $\mathbb{A} = \{a_1, \dots, a_n\}$ be an alphabet and p be the Parikh image function. That is, the function $p : \mathbb{A}^* \to \mathbb{N}^n$ that associates each word with a vector that counts the number of appearances of each letter, $p(w)(i) = |\{j \mid w(j) = a_i\}|$, where $w(j)$ denotes the jth letter of the string w, starting at 1.

Definition 3.7. We say that a regular language $\mathcal{L}$ is **extensible** if for every $m \in \mathbb{N}$ and word $w \in \mathcal{L}$ there exists another word $w' \in \mathcal{L}$ such that for any coordinate $i \in [n]$,

$$\text{if } p(w)(i) \neq 0 \text{ then } p(w')(i) \geq m, \text{ and}$$
$$\text{if } p(w)(i) = 0 \text{ then } p(w')(i) = 0.$$

In this case we say that w' is an **m-extension** of w. We write $\mathscr{E}$ for the class of all extensible regular languages. Likewise we define a *tree* language to be extensible if it can be defined by an unranked tree automaton whose transitions only use languages from $\mathscr{E}$ to test properties on the siblinghoods. Note that this is a restriction of the

[5]For example, $(\exists v.D_1(v)) \wedge \neg(\exists v.D_2(v) \wedge D_3(v))$ is bounded by 2, since there are at most 2 D_i's used in each quantified subformula.

horizontal tests, and that an extensible tree language can still test for any regular property along a branch. By $\mathscr{E}_{tree}$ we denote the set of extensible tree languages.

Let us give some examples of extensible classes of languages. As a first example, consider the following class of languages.

$\exists$-*class* $:= \{\mathcal{L}_{\mathbb{A}_1,\ldots,\mathbb{A}_n,\mathbb{B}} \mid \mathbb{A}_1,\ldots,\mathbb{A}_n$ and $\mathbb{B}$ are a finite alphabets, $n \in \mathbb{N}\}$, where $\mathcal{L}_{\mathbb{A}_1,\ldots,\mathbb{A}_n,\mathbb{B}} := \{w \in (\cup_i \mathbb{A}_i \cup \mathbb{B})^* \mid w$ contains at least one letter from each $\mathbb{A}_i\}$

Note that $\exists$-*class* is a class of extensible languages, it is closed under intersection, complementation, inverse homomorphisms and contains all membership languages. Therefore, the class of data tree languages accepted by $\exists$-*class*-DD automata is closed under complementation. If we also close $\exists$-*class* under componentwise product, we still obtain an extensible language. In this case, the class of languages accepted by DD automata is closed by all boolean operaitons.

Also note that the class of $\exists$-*class*-transducers are those that can only test for the existence or non existence of children with a certain label, but they do not test any condition on the horizontal ordering or on the number of appearances of elements in the siblinghoods. In fact, they can mostly test for *vertical* properties along branches.

As another example, we define REG_* the class of *star*-regular languages.

$$REG_* := \{\mathcal{L} \in REG \mid \mathcal{L} \text{ is defined by a regular expression whose every symbol is in the scope of at least one } *\}$$

Thus, for example '$((a \mid b)\, c^* d)^*$' is in REG_* while '$a^* b$' is not. The class REG_* is trivially extensible.

The property of extensibility is trivially closed under union, but not necessarily under intersection or complementation.

PROPOSITION 3.8. *Given two extensible languages $\mathcal{L}_1, \mathcal{L}_2$, $\mathcal{L}_1 \cup \mathcal{L}_2$ is also extensible.*

As a counter-example for the intersection, note that $(ab)^* \cap a^* b^* = \{\epsilon, ab\}$ which is not extensible, and for the complementation note also that under the singleton alphabet $\mathbb{A} = \{a\}$, $(a^+ a)^c = \{\epsilon, a\}$, which is not extensible.

In the sequel, we will show decidability of the emptiness problem for $\mathscr{E}$-DD automata.

4. THE EMPTINESS PROBLEM

Throughout this section, we fix the transducer to be an $\mathscr{E}$-transducer. The main objective of this section is to prove the following theorem.

THEOREM 4.1. *The emptiness problem for $\mathscr{E}$-Downward Data automata can be decided in* 2EXPTIME.

Sketch of the proof

The proof of the main theorem is divided into four parts. In the first part (Section 4.1) we define some decoration or marking of the nodes of a data trees that in some sense witnesses the acceptance of a run of a DD automaton. These decorations of the tree are the main structures with which we will work with in our proof.

The second part (Section 4.2) is dedicated to proving two properties. The first property states that if a DD automaton is nonempty, it accepts a tree decorated with some guidance system that marks the paths to be covered in order to verify the properties imposed by the verifier. In some sense, it decorates the tree as in Figure 2, avoiding having two paths going through the same node. The guidance system is called *certificate* and this property is called *admissibility of correct certificates.* The second property states that if a DD automaton is nonempty, then it accepts a tree whose data values are in a certain normal form, as follows. Every pair of subtrees rooted at two different children of a node, have a disjoint set of data values, with the exception of some polynomially bounded many (this is called the *disjoint values property*).

The third part (Section 4.3) is centered around proving that DD automata have the exponential width model property. That is, if a DD automaton has a nonempty language, then it accepts a tree whose width is exponentially bounded in the size of the automaton.

In the fourth part (Section 4.4) we give the algorithm for testing emptiness of DD automata, which is based on the bound on the width and the two previous properties: the disjoint values property and the admissibility of correct certificates.

Parameters

We first fix some parameters that we will need in the complexity analysis. We fix, once and for all, that the verifier $\mathcal{V}$ contains K NFA: $\mathcal{A}_1, \dots, \mathcal{A}_\mathsf{K}$. We write Aut to denote the se of these automata, $Aut = \{\mathcal{A}_1, \dots, \mathcal{A}_\mathsf{K}\}$. We assume without any loss of generality that all $\mathcal{A}_1, \dots, \mathcal{A}_\mathsf{K}$ share the same set of states $\mathcal{Q} := \{q_1, \dots, q_\mathsf{N}\}$, and have q_1 as initial state. Also, for each i we write $\mathcal{Q}_F^{\mathcal{A}_i}$ for the set of final states of $\mathcal{A}_i$. By $|\mathcal{A}|$ we denote the number of transitions of $\mathcal{A}$, and Aut stands for $|\mathcal{A}_1| + \cdots + |\mathcal{A}_\mathsf{K}|$. Let $Vars$ be the set of variables used by the formulæ of the verifier, and let $|Vars| = \mathsf{V}$. We write R for the maximum number of relations admitted under a quantification. In other words, for any formula of the verifier and for any quantified subformula $\exists \bar{x}.\psi$, there are at most R different relations used in ψ from the K available. The worst case would be when $\mathsf{R} = \mathsf{K}$, as in the formula $\mathsf{v}(b) = \exists x.D_1(x) \wedge \cdots \wedge D_\mathsf{K}(x)$ for some $b \in \mathbb{B}$. This is an important parameter, since we will later see that the subclass of $\mathscr{E}$-DD automata with R fixed has an ExpTime emptiness problem, while the general class has a 2ExpTime emptiness problem. In Section 6.1 we will argue that downward XPath expressions can be translated into $\mathscr{E}$-DD automata where $\mathsf{R} = 2$, and from this fact it will follow that its satisfiability is in ExpTime. Intuitively, the class of properties that do not have a bounded R are of the form *there is a data value d accessible by n downward paths in the regular languages $\mathcal{L}_1, \dots, \mathcal{L}_n$ but not by any path in the languages $\mathcal{L}'_1, \dots, \mathcal{L}'_m$*, where n and m are parameters. To verify these properties with our approach would require double exponential time in n, m.

Finally, we fix $\dot{Q}$ to denote the set of states of the transducer. In addition, we use NFA to represent the regular language $\mathcal{L}$ for every transition $(\dot{q}, a, b, \mathcal{L}) \in \delta$. We will usually use the symbol $\mathcal{B}$ to denote such an automaton. We will assume without any loss of generality that all automata used in the transitions of the transducer share the same set of states $\tilde{Q} = \{\tilde{q}_0, \tilde{q}_1, \dots\}$ and that all automata have the same initial state $\tilde{q}_0$. $|\mathcal{R}|$ stands for the number of transitions of $\mathcal{R}$, and we assume that $|\dot{Q}|$ is at most $|\mathcal{R}|$. By $|\mathcal{V}|$ we denote $\mathsf{K} + \mathsf{N} + \mathsf{R} + \mathsf{V} + \mathsf{Aut}$. Summing up, our complexity analysis will be based on the parameters: $\mathsf{K}, \mathsf{N}, \mathsf{R}, \mathsf{V}, |\dot{Q}|, |\tilde{Q}|, |\mathcal{R}|, \mathsf{Aut}, |\mathcal{V}|$.

4.1 Decorations of trees

In order to bound the width of the tree, we need a more fine grained notion of runs of a transducer. We label the nodes of the tree with the run of the automaton recognizing the regular language on the siblinghood used at the transition of the transducer's run. This will enable us to state a pumping argument on siblinghoods of the data tree.

Detailed run

Consider, for any extensible language $\mathcal{L} \in \mathscr{E}$ over $\dot{Q}$, a NFA $\mathcal{B}_{\mathcal{L}}$ with initial state $\tilde{q}_0$. Remember that every $\mathcal{B}_{\mathcal{L}}$ used in the transducer uses the same set $\tilde{Q}$ of states.

Definition 4.2. A **detailed run of a transducer** $\mathcal{R}$ on a tree $\mathbf{a} \otimes \mathbf{b} : P \to \mathbb{A} \times \mathbb{B}$ consists of a 3-uple (τ, ρ, ρ_h).

—ρ is a run, that in this context we call a *vertical run.*

—τ is the *transition assignment*

$$\tau : P \to \delta$$

specifying which choice of transitions are needed for the run ρ. It verifies, for every position x with l children,

$$\tau(x) = (\rho(x), \mathbf{a}(x), \mathbf{b}(x), \mathcal{L}) \text{ where } \rho(x{\cdot}1) \cdots \rho(x{\cdot}l) \in \mathcal{L}.$$

We abbreviate $\mathcal{B}_x$ to denote the NFA $\mathcal{B}_{\mathcal{L}}$ of the regular language $\mathcal{L}$ defined in the transition $\tau(x)$ of the transducer's run.

—Finally, ρ_h is an assignment from the set of positions P to the set of states of the automaton $\mathcal{B}$ that corresponds to the regular language that needs to be checked in order to apply the transition.

$$\rho_h : P \to \tilde{Q} \ .$$

We call ρ_h the *horizontal run.* It verifies the following conditions (in short, that it is a run of a NFA on the siblinghood).

1. For every leftmost sibling $x{\cdot}1 \in P$, $(\tilde{q}_0, \rho(x{\cdot}1), \rho_h(x{\cdot}1))$ is a transition of $\mathcal{B}_x$.
2. For every pair of consecutive siblings $x{\cdot}i, x{\cdot}(i+1) \in P$,

$$(\rho_h(x{\cdot}i), \rho(x{\cdot}(i+1)), \rho_h(x{\cdot}(i+1)))$$

is a transition of $\mathcal{B}_x$.

3. For every rightmost sibling $x{\cdot}l$, $\rho_h(x{\cdot}l)$ is a final state of $\mathcal{B}_x$.

This completes the definition of a detailed run.

We also need to be able to precisely describe the behavior of the *data values* at a position of a data tree.

Description of data

Next we introduce sets of data simultaneously accessible by R automata (remember that this is the maximum number of simultaneous relations D_i's used by the verifier $\mathcal{V}$). For each one of these sets of data values, we preserve *at most* V elements (this is a bound on the maximum number of variables used by $\mathcal{V}$).

Observe that the verifier can test properties of the set of the cardinality of the sets $[\![\mathcal{A}_i]\!]$, or a boolean combination of them. More precisely, the verifier can only test the (in)existence of a number of data values that are in some *intersection* set $[\![\mathcal{A}_{i_1}]\!] \cap \cdots \cap [\![\mathcal{A}_{i_t}]\!]$. Here, $\mathcal{A}_{i_1}, \dots, \mathcal{A}_{i_t}$ range over Aut, and $t \leq \mathsf{R}$. The tests boil down to checking whether

—the intersection contains *at least* $1, 2, \dots, \mathsf{V}$ different elements, or

—the intersection contains *at most* $0, 1, \dots, \mathsf{V} - 1$ different elements.

Note that we cannot test, for example, that the set contains *exactly* V elements, since we would need a formula with $\mathsf{V} + 1$ variables. We annotate the tree with this information. Since later we will need to check that this information is consistent between a parent position and its children, we need to also consider the states of the automata $\mathcal{A}_i$. Next, we define the set of intersections of at most R relations $[\![\mathcal{A}_{i_1}, q_{j_1}]\!] \cap \cdots \cap [\![\mathcal{A}_{i_\mathsf{R}}, q_{j_\mathsf{R}}]\!]$.

$$Inters := \wp_{\leq \mathsf{R}}(Aut \times \mathcal{Q}) \ ,$$

where $\wp_{\leq \mathsf{R}}$ denotes the set of subsets of at most R elements. The following holds by definition.

$$|Inters| \leq (\mathsf{K} \cdot \mathsf{N})^{\mathsf{R}} \tag{1}$$

Given a data tree $\mathbf{t}$ and an intersection we extend the $[\![\ ]\!]^{\mathbf{t}}$ notation by taking the intersection of the R sets. That is,

$$[\![I]\!]^{\mathbf{t}} = \bigcap \{ [\![\mathcal{A}, q]\!]^{\mathbf{t}} \mid (\mathcal{A}, q) \in I \} \ , \quad \text{for } I \in Inters.$$

Definition 4.3. The **data profile** is a function that assigns the number of elements present at each $I \in Inters$ to every data tree as follows.

$$\begin{aligned} d\text{-}profile &: Trees(\mathbb{A} \times \mathbb{D}) \to Inters \to [0..\mathsf{V}] \\ d\text{-}profile(\mathbf{t}) &= \{ I \mapsto \min(|[\![I]\!]^{\mathbf{t}}|, \mathsf{V}) \mid I \in Inters \} \end{aligned}$$

Observe that since V is the number of variables in Φ, it is enough to count up to V.

A tree's profile carries sufficient information to evaluate at the root any formula $\varphi \in \Phi$ used by the verifier. We write $f \models \varphi$ if f is a function $f : Inters \to [0..\mathsf{V}]$ and φ is a formula of the verifier $\mathsf{v}(b) = \varphi$ such that φ holds in a tree $\mathbf{t}$ if $f = d\text{-}profile(t)$. We formally define this relation in Table I. Thus, for any data tree $\mathbf{t}$, position x, and $\varphi \in \Phi$, $d\text{-}profile(\mathbf{t}|_x) \models \varphi$ iff φ holds at $\mathbf{t}|_x$.

A profile summarizes information about a position in terms of data values in its subtrees. In addition, we also define the description of a data value in terms of the different ways by which it can be obtained.

$$f \models \psi \wedge \psi' \text{ iff } f \models \psi \text{ and } f \models \psi'$$
$$f \models \neg\psi \text{ iff } f \not\models \psi$$
$$f \models \exists v_1, \ldots, v_t . \psi \text{ iff } I, g, h \models \psi \text{ for some } I \in \mathit{Inters},\ g : \{v_1, \ldots, v_t\} \to \{1, \ldots, t\}$$
$$\text{and } h : \{1, \ldots, t\} \to \wp(I) \text{ s.t. } |h^{-1}(I')| \leq f(I') \text{ for all } I' \subseteq I$$

$$I, g, h \models \psi \wedge \psi' \text{ iff } I, g, h \models \psi \text{ and } I, g, h \models \psi' \qquad I, g, h \models v = v' \text{ iff } g(v) = g(v')$$
$$I, g, h \models \psi \vee \psi' \text{ iff } I, g, h \models \psi \text{ or } I, g, h \models \psi' \qquad I, g, h \models v \neq v' \text{ iff } g(v) \neq g(v')$$
$$I, g, h \models D_i(v) \text{ iff } (\mathcal{A}_i, q_1) \in h(g(v))$$

Table I. The relation $f \models \varphi$ given $f : \mathit{Inters} \to [0..\mathsf{V}]$.

Definition 4.4. The **description** of a data value is the set of states of the automata that can access the data value and it is defined as

$$\mathrm{desc}_{\mathbf{t}}(d) := \{(\mathcal{A}, q) \in \mathit{Aut} \times \mathcal{Q} \mid d \in [\![\mathcal{A}, q]\!]^{\mathbf{t}}\} \in \mathit{Descriptions}, \text{ where}$$
$$\mathit{Descriptions} := \wp(\mathit{Aut} \times \mathcal{Q}).$$

Certificates

For any intersection I, we want to keep track of *which* data values are in $[\![I]\!]$, and *how to access* them in the subtree in order to verify that they belong to every $[\![\mathcal{A}, q]\!]$ in I. How do we decorate the tree in order to have this information at all times? Suppose $\mathbf{t}$ is a data tree and x a position in it. We will develop some branch marking system. For $d \in [\![I]\!]^{\mathbf{t}}$, we mark several downward paths starting in x and ending at a lower positions $y \succeq x$ with $\mathbf{d}(y) = d$. We do this in such a way that for every $(\mathcal{A}, q) \in I$ there is a marked path between x and y such that $\mathbf{d}(y) = d$ and $q \xrightarrow[x,y]{\mathcal{A}} q_f$ with q_f a final state. We will mark every element of this path with the data value 'd' to which it leads. We call this marking a *certificate.* However, markings of paths should not overlap. That we can always have such non-overlapping certificates is not obvious, and it will be the matter of Section 4.2.

A **certificate** of a data tree $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$ is a partial assignment $\kappa : \mathsf{pos}(\mathbf{t}) \rightharpoonup \mathit{data}(\mathbf{t})$. If κ is undefined for a position x, we write $\kappa(x) = \bot$, and we extend the $\mathrm{desc}_{\mathbf{t}}$ function with $\mathrm{desc}_{\mathbf{t}}(\bot) := \emptyset$ for convenience in the proofs. A certificate κ is said to be *correct* if it has the properties of being *valid* and *inductive* that we will define next.

The **validity** property for a position x ensures that the certificate takes into account all the necessary data values to witness every intersection. That is, that for every intersection I the data values of $[\![I]\!]^{\mathbf{t}|_x}$ are contained either in $\kappa(x)$ or in some $\kappa(x \cdot i)$ for a child position $x \cdot i$ of x. Since the verifier has only V variables, it is actually sufficient to verify the existence of certificates for up to V data values from $[\![I]\!]^{\mathbf{t}|_x}$. This property, when combined with the inductive property results in each of these data values having a path of certificates that witness each of the elements of I.

Definition 4.5. Given $I \in \mathit{Inters}$, $D \subseteq \mathbb{D}$, $C \subseteq \mathbb{D} \times \mathit{Descriptions}$, and given

$d_{cert}, b_{curr}, d_{curr} \in \mathbb{D}$, then

$$valid(\, D,\, I,\, (b_{curr}, d_{curr}, d_{cert}),\, C\,)$$

holds if there are $k = \min(\mathsf{V}, |D|)$ different data values $d_1, \ldots, d_k \in D$ such that for every variable $i \in [k]$ and $(\mathcal{A}, q) \in I$, there exists $(q, b_{curr}, q') \in \mathcal{A}$ and

—$q' \in Q_F^{\mathcal{A}}$ and $d_{curr} = d_{cert} = d_i$, or

—there is $(d_i, Desc) \in C$, with $(\mathcal{A}, q') \in Desc$.

The **inductivity** property states that for every position x such that $\kappa(x) = d \neq \perp$, if d is in some $[\![\mathcal{A}, q]\!]$, then there must exist a child position $x{\cdot}i$ with certificate d such that d is in $[\![\mathcal{A}, q']\!]^{\mathbf{t}|_{x \cdot i}}$ for some q' in the transition relation of $\mathcal{A}$.

Definition 4.6. Given $Desc_{cert} \in Descriptions$, then

$$inductive(\, Desc_{cert},\, (b_{curr}, d_{curr}, d_{cert}),\, C\,)$$

holds iff for every $(\mathcal{A}, q) \in Desc_{cert}$, there is $(q, b_{curr}, q') \in \mathcal{A}$ such that

—$q' \in Q_F^{\mathcal{A}}$ and $d_{cert} = d_{curr}$, or

—there is $(d_{curr}, Desc) \in C$ with $(\mathcal{A}, q') \in Desc$.

Definition 4.7. A certificate κ is **correct** if for every position $x \in \mathsf{pos}(t)$ there exists a subset of children $\mathcal{C}_x \subseteq \{x{\cdot}i \mid x{\cdot}i \in \mathsf{pos}(\mathbf{t})\}$ such that

$$inductive(\ \mathrm{desc}_{\mathbf{t}|_x}(\kappa(x)),\ (\mathbf{b}(x), \mathbf{d}(x), \kappa(x)),\ \hat{\mathcal{C}}_x\ )$$

holds, where $\hat{\mathcal{C}}_x = \{(\kappa(y), \mathrm{desc}_{\mathbf{t}|_y}(\kappa(y))) \mid y \in \mathcal{C}_x\}$; and for every intersection $I \in Inters$ the valid property holds,

$$valid(\ [\![I]\!]^{\mathbf{t}|_x},\ I,\ (\mathbf{b}(x), \mathbf{d}(x), \kappa(x)),\ \hat{\mathcal{C}}_x\ )\ .$$

In this context we say that $\hat{\mathcal{C}}_x$ is a valid and inductive subset of children positions of x.

Take any x and $(\mathcal{A}, q) \in I \in Inters$ with $k = \min(\mathsf{V}, [\![I]\!]^{\mathbf{t}|_x})$. The correctness condition implies that

—there are $d_1, \ldots, d_k$ different data values and $x_1, \ldots, x_k \in \mathsf{pos}(\mathbf{t})$ below x such that for all i: $\mathbf{d}(x_i) = d_i$;

—for all $x \prec y \preceq x_i$, $\kappa(y) = d_i$; and

—$q \xrightarrow[x, x_i]{\mathcal{A}} q_f$ for some $q_f \in Q_F^{\mathcal{A}}$.

Note that not every data tree accepted by a $\mathscr{E}$-DD automaton has a correct certificate. (Think for instance in a tree with branching width 1, in which in order to verify the root's property, two different data values are needed.) Indeed, admissibility of correct certificates is a property shared only by *some* of the trees recognized by a $\mathscr{E}$-DD automaton. However, in Section 4.2 we will show that every nonempty $\mathscr{E}$-DD automaton accepts a tree which admits a correct certificate. Even more, we show that it accepts a tree where additionally the data values have a particular property, that we define as the *disjoint values property.*

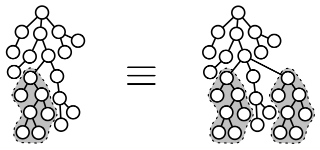

Fig. 4. Subtree replication.

4.2 Correct certificates and disjoint values

This section is dedicated to proving two central properties that are essential to obtain a decidability procedure for the DD automata emptiness problem. These properties state that every nonempty DD automaton accepts a tree that (1) admits a correct certificate, and (2) has the disjoint values property —a property that we will define in Section 4.2.2.

Firstly, in Section 4.2.1 we attack the question of whether we can always assume that we have a *correct* certificate, which is a property that is not shared by all runs. The next section is devoted to showing that for any tree $\mathbf{t}$ accepted by a DD automaton there exists a transformation of this tree $\mathbf{t}'$ obtained by duplicating some subtrees, that is also accepted by the automaton.

Secondly, Section 4.2.2 treats the question of whether we can always assume that the run and certificate satisfy the disjoint values property. We will show that given a correctly certified data tree, we can always rearrange the data values in order to meet this property, while preserving the run and the certificate.

We define the function $\hat{\kappa}$ by

$$\hat{\kappa}(x) := \{\kappa(x{\cdot}i) \mid x{\cdot}i \in \mathsf{pos}(\mathbf{t})\} \cup \{\kappa(x)\},$$

that is, $\hat{\kappa}$ assigns to every position x, the set of data values of the certificates of the children of x, as well as of x.

The next two sections will show that the following theorem holds.

Theorem 4.8. *For every $\mathscr{E}$-DD automaton $(\mathcal{R}, \mathcal{V})$ that accepts a non-empty language, there is a tree $\mathbf{t}$ and a correct certificate of $\mathbf{t}$ with the disjoint values property such that $\mathbf{t}$ is accepted by $(\mathcal{R}, \mathcal{V})$.*

4.2.1 *Correct certificates.* In this section we exploit the particular property of *extensibility* (Definition 3.7) that we imposed to the regular languages used by $\mathscr{E}$-transducers.

As in previous sections, we have that every verifier we consider has a maximum number of relations inside a quantified subformula bounded by R, maximum number of variables bounded by V, the number of automata is K, and the number of states in any automaton is bounded by N.

Our first observation is that if we duplicate a subtree of a data tree as in Figure 4, the values of the certificates do not change, and either both trees are accepted or both rejected by any verifier. The following easy proposition is given without a proof.

Proposition 4.9. *Given a data tree $\mathbf{t}$, and given a position $x{\cdot}i \in \mathsf{pos}(\mathbf{t})$ consider the last index l such that $x{\cdot}l \in \mathsf{pos}(\mathbf{t})$. Let $\mathbf{t}'$ be the follwing data tree that* *results from duplicating the subtree* $\mathbf{t}|_{x\cdot i}$ *in* $\mathbf{t}$*. We define* $\mathbf{t}' := (\mathbf{t} \circ f_x)$*, where* f_x *is the following surjective function.*

$$f_x : \mathsf{pos}(\mathbf{t}) \cup \{x\cdot(l+1)\cdot y \mid x\cdot i\cdot y \in \mathsf{pos}(\mathbf{t})\} \to \mathsf{pos}(\mathbf{t})$$

$$f_x(z) = \begin{cases} z & \text{if } z \not\succeq x\cdot(l+1) \\ x\cdot i\cdot y & \text{if } z = x\cdot(l+1)\cdot y \end{cases}$$

Then, for every $z \in \mathsf{pos}(\mathbf{t}')$*, and* $d \in \mathbb{D}$*,* $desc_{\mathbf{t}'|_z}(d) = desc_{\mathbf{t}|_{f_x(z)}}(d)$*.*

This can be also extended to $\mathcal{R}$-transducer runs, always by replicating subtrees. Here, the *extensibility* of the regular languages of $\mathcal{R}$ will be of utmost importance.

Proposition 4.10. *If we have*

—*a* $\mathscr{E}$*-DD automaton* $(\mathcal{R}, \mathcal{V})$*,*
—*a data tree* $\mathbf{t} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$*,*
—*a position* $x \in \mathsf{pos}(\mathbf{t})$ *with* $l = \#children(\mathbf{t}, x)$*,*
—*a run* ρ *of* $\mathcal{R}$ *on* $\mathbf{t}$
—*a regular language* $\mathcal{L}$ *such that* $(\rho(x), \mathbf{a}(x), \mathbf{b}(x), \mathcal{L}) \in \delta$ *and* $\rho(x\cdot 1) \cdots \rho(x\cdot l) \in \mathcal{L}$
—*an extension* $\dot q_1 \cdots \dot q_n \in \mathcal{L}$ *of* $\rho(x\cdot 1) \cdots \rho(x\cdot l)$ *(where of course* $n > l$*), and any* surjective *function* $h_x : [n] \to [l]$ *such that* $\dot q_i = \rho(x\cdot h_x(i))$ *for every* $i \in [n]$*,*
—*a data tree defined as* $\mathbf{t}' := (\mathbf{t} \circ f_x)$*, whose set of positions is*

$$P = \{x\cdot j\cdot y \mid x\cdot h_x(j)\cdot y \in \mathsf{pos}(\mathbf{t})\} \cup \{z \mid z \in \mathsf{pos}(\mathbf{t}), x \not\prec z\}$$

and where f_x *is as follows.*

$$f_x : P \to \mathsf{pos}(\mathbf{t})$$

$$f_x(y) = \begin{cases} y & \text{if } x \not\prec y \\ x\cdot h_x(j)\cdot y & \text{if } y = x\cdot j\cdot y \end{cases}$$

Then:

(1) For every position $y \in \mathsf{pos}(\mathbf{t}')$ *and data value* d*,* $desc_{\mathbf{t}'|_y}(d) = desc_{\mathbf{t}|_{f_x(y)}}(d)$*.*
(2) $\rho \circ f_x$ *is a run of* $\mathcal{R}$ *on* $\mathbf{t}'$*.*

The proof for this Proposition can be found in the online appendix.

Remark 4.11. Note that Proposition 4.10 implies that if $\mathbf{t}$ is accepted by a verifier, then $\mathbf{t}'$ is also accepted, and idem for the transducer.

Now we can show that we can always restrict to *correct* certificates.

Proposition 4.12. *Every nonempty* $\mathscr{E}$*-DD automaton accepts some data tree with a* correct *certificate.*

Proof. Let us fix a DD automaton $(\mathcal{R}, \mathcal{V})$. We will show the following statement.

Claim 4.13. *Given a data tree* $\mathbf{t} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$*, a run* ρ *on* $\mathbf{t}$*, and a data value* $e \in data(\mathbf{t}) \cup \{\bot\}$ *such that* $\mathbf{t}$ *is accepted by* $\mathcal{V}$*, there exists another tree* $\mathbf{t}'$ *with the same height as* $\mathbf{t}$*, with run* ρ' *and a* correct *certificate* κ' *such that* $\kappa'(\epsilon) = e$*,* $\rho'(\epsilon) = \rho(\epsilon)$ *and* $\mathbf{t}'$ *is accepted by* $\mathcal{V}$ *according to* ρ'*. Further, for every* $d \in \mathbb{D}$*,* $desc_{\mathbf{t}}(d) = desc_{\mathbf{t}'}(d)$*.*

Since $(\mathcal{R}, \mathcal{V})$ is nonempty, there exists a data tree $\mathbf{t}$ with an accepting run ρ. If we take $e = \bot$, by Claim 4.13 we obtain a data tree $\mathbf{t}'$ with an accepting run ρ', and a correct certificate κ', proving the statement of the proposition.

Proof of Claim 4.13. We proceed by induction on the height of $\mathbf{t}$.

The base case consists in showing that the property holds for a tree $\mathbf{t}$ of height 0 consisting of only one position: ϵ. In this case it suffices to define $\mathbf{t}' = \mathbf{t}$, $\kappa'(\epsilon) = e$ and $\rho' = \rho$, and all the properties are trivially met.

For the inductive case, suppose that for all trees of height at most h the statement holds. Let $\mathbf{t}$ be of height $h+1$ with a run ρ and let $e \in \mathit{data}(\mathbf{t}) \cup \{\bot\}$. We need to provide a tree $\mathbf{t}'$ run ρ' and certificate κ' with the desired properties.

The basic idea is to build $\mathbf{t}'$ by replicating some subtrees of $\mathbf{t}$ to allow to have sufficently non-overlaping paths to generate the correct certificate. To build such tree and certificate, it is useful to have a function that, given a data value d, a NFA $\mathcal{A}$ and a state q, returns a position x with data value d such that $\mathrm{str}(\epsilon, x) \in L(\mathcal{A}[q])$. If there is no such witnessing position, (*i.e.*, if $d \notin [\![\mathcal{A}, q]\!]^{\mathbf{t}}$) it returns an undefined value. We denote by posWitness any such function. It has the next property, for every data value d, automaton $\mathcal{A}$ and state q.

—$\mathsf{posWitness}(d, \mathcal{A}, q) \in \{y \in \mathsf{pos}(\mathbf{t}) \mid \mathrm{str}(\epsilon, y) \in L(\mathcal{A}[q]),\ \mathbf{d}(y) = d\}$, or

—$\mathsf{posWitness}(d, \mathcal{A}, q) = \bot$ if $\{y \in \mathsf{pos}(\mathbf{t}) \mid \mathrm{str}(\epsilon, y) \in L(\mathcal{A}[q]),\ \mathbf{d}(y) = d\} = \emptyset$.

Remember that the certificate we build κ' need to have the data value e at the root. In order to verify the *inductivity* condition at the root, we will collect every witness position for the data value e. In some sense we want to gather information about which subtrees $\mathbf{t}|_j$ need to have a certificate with data value e.

$$A = \{(e, j) \mid j \preceq \mathsf{posWitness}(e, \mathcal{A}, q), \mathcal{A} \in \mathit{Aut}, q \in \mathcal{Q}, j \in \mathsf{pos}(\mathbf{t})\}$$

In order to build a *valid* certificate at the root, we also must take into account the subtrees $\mathbf{t}|_j$ that are necessary to verify the formulæ for every intersection I.

$$\mathcal{C}_I = \{(d, j) \mid d \in [\![I]\!]^{\mathbf{t}}, j \preceq \mathsf{posWitness}(d, \mathcal{A}, q), (\mathcal{A}, q) \in I, j \in \mathsf{pos}(\mathbf{t})\}$$

We must then witness all the elements of $E = A \cup \bigcup\{\mathcal{C}_I \mid I \in \mathit{Inters}\}$. We build a data tree $\mathbf{t}''$, which is the result of duplicating some of the subtrees of $\mathbf{t}$ so as to have enough space to fit all the necessary witness certificates required by E. We then need at most $|E|$ replications of trees. This is achieved by the *extensibility* of the languages corresponding to the transitions of ρ. Let us consider $\dot{q}_1, \dots, \dot{q}_n$ to be an $|E|$-extension of $\rho_i(1) \cdot \cdots \cdot \rho_i(l) \in \mathcal{L}$ for a language $\mathcal{L}$ such that $(\rho(\epsilon), \mathbf{a}(\epsilon), \mathbf{b}(\epsilon), \mathcal{L}) \in \delta$. We define f_ϵ as in Proposition 4.10, and we define $\mathbf{t}'' = \mathbf{t} \circ f_\epsilon$, $\rho'' = \rho \circ f_\epsilon$. It follows that ρ'' is a run on $\mathbf{t}''$ and by Remark 4.11 $\mathbf{t}''$ is accepted by $\mathcal{V}$.

To define κ', we must make sure that for each element of E there is a corresponding certificate in some child of the root of $\mathbf{t}''$. Of course, there cannot be two certificates in the same child, so we need a way to choose distinct children for distinct elements of E. Let g be a function that chooses, for each element of E, which subtree to use, $g : E \to \{j \mid j \in \mathsf{pos}(\mathbf{t}'')\}$ such that g is injective and $f_\epsilon(g(d, j \cdot y)) = j$ for every $(d, j \cdot y) \in E$.

We are in a good shape to define a certificate κ' for $\mathbf{t}''$ that satisfies the correctness property at the root. But we need κ' to be correct also at all subtrees of $\mathbf{t}''$. This is given by the inductive hypothesis.

For every subtree $\mathbf{t}''|_i$ of $\mathbf{t}''$ we apply the inductive hypothesis with the data value d_0 such that $g^{-1}(i) = (d_0, y)$ for some y, or with $\perp$ otherwise. We thus obtain a tree $\mathbf{t}'_i$ with a correct certificate κ'_i and run ρ'_i. Finally we build the following tree $\mathbf{t}'$, certificate κ', and run ρ' satisfying all the properties.

$$\mathbf{t}'(x) = \begin{cases} \mathbf{t}(x) & \text{if } x = \epsilon \\ \mathbf{t}'_i(y) & \text{if } x = i{\cdot}y,\ i \in \mathsf{pos}(\mathbf{t}''),\ y \in \mathsf{pos}(\mathbf{t}'_i) \end{cases}$$

$$\kappa'(x) = \begin{cases} e & \text{if } x = \epsilon \\ \kappa'_i(y) & \text{if } x = i{\cdot}y,\ i \in \mathsf{pos}(\mathbf{t}''),\ y \in \mathsf{pos}(\mathbf{t}'_i) \end{cases}$$

$$\rho'(x) = \begin{cases} \rho(x) & \text{if } x = \epsilon \\ \rho'_i(y) & \text{if } x = i{\cdot}y,\ i \in \mathsf{pos}(\mathbf{t}''),\ y \in \mathsf{pos}(\mathbf{t}'_i) \end{cases}$$

ρ' is a run on $\mathbf{t}'$ since it is composed of runs ρ'_i from the subtrees and at the root it satisfies the transition $(\rho'(\epsilon), \mathbf{a}(\epsilon), \mathbf{b}(\epsilon), \mathcal{L}) \in \delta$. κ' is a correct certificate, since the κ_i's are correct certificates for all the subtrees, and κ verifies the inductive and valid conditions at the root. The description of any data value d at the root was not altered since we only applied Proposition 4.10 and the inductive hypothesis that preserve the descriptions. Finally, we can see that $\mathcal{V}$ accepts $\mathbf{t}'$ since it accepts each of its subtrees, and also has the property of the root, since no descriptions of data values were modified. $\square$

4.2.2 *Disjoint values.* We introduce a property concerning the data values of the tree. The idea is that given two disjoint subtrees $\mathbf{t}|_x$, $\mathbf{t}|_y$ with $x \not\preceq y$, $y \not\preceq x$, the only data values they can share, if any, are those of the certificates of their roots $\kappa(x), \kappa(y)$, or those of some of their children $\kappa(x{\cdot}i), \kappa(y{\cdot}j)$. Remember that these last ones constitute all the witness data that are necessary to verify the profile at x and y. All other data values can be assumed to occur in only one of the two subtrees. Here we show that for every nonempty DD automaton there is always a tree that can be certified in such a way that this property holds. Next we formalize the *disjoint values property*, which will be an essential property in order to prove our main decidability result of Theorem 4.1.

Definition 4.14. Let $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$ be a data tree recognized by a DD automaton $(\mathcal{R}, \mathcal{V})$. Let κ be a correct certificate. We say that it has the **disjoint values** property if, for every pair of incomparable positions $x, y \in \mathsf{pos}(\mathbf{a} \otimes \mathbf{d})$,

$$\mathit{data}(\mathbf{t}|_x) \cap \mathit{data}(\mathbf{t}|_y) \quad \subseteq \quad \hat{\kappa}(x) \cap \hat{\kappa}(y)\ .$$

We show here that we can always assume the model to have the disjoint values property. The idea is that once we have a correct certificate κ over a data tree $\mathbf{t}$, we know that at any inner node $x \in \mathsf{pos}(\mathbf{t})$, all the important data values to verify *d-profile*$(\mathbf{t}|_x)$ shared between subtrees $\{\mathbf{t}|_{x \cdot i} \mid x{\cdot}i \in \mathsf{pos}\}$, are those contained in the certificates of the children $\kappa(x{\cdot}i)$ or $\kappa(x)$. That is, for every x, the only necessary data values to verify its profile (or an ancestor's profile) is in $\hat{\kappa}(x)$. We can then ensure that these are the only data values that may be shared by any two $\mathbf{t}|_{x\cdot i}$, $\mathbf{t}|_{x\cdot j}$.

We now state the important proposition of this section.

PROPOSITION 4.15 (DISJOINTNESS). *Given*

—*a data tree* $\mathbf{t}$ *with a correct certificate* κ*, and* $x \in \mathsf{pos}(\mathbf{t})$*,*

—*a data value* $d_x \in data(\mathbf{t}|_x) \setminus \hat{\kappa}(x)$*,* $d'_x \notin data(\mathbf{t})$*,*

—*the bijective function* $f : \mathbb{D} \to \mathbb{D}$ *such that* $f(d) = d$ *for all* $d \in data(\mathbf{t}) \setminus \{d_x\}$ *and* $f(d_x) = d'_x$*,*

—*the data tree* $\mathbf{t}'$ *with* $\mathsf{pos}(\mathbf{t}') = \mathsf{pos}(\mathbf{t})$ *and a certificate* κ'*, defined as follows.*

$$\mathbf{t}'(y) = \begin{cases} (\mathbf{a}(y), (f \circ \mathbf{d})(y)) & \text{if } y \succeq x \\ \mathbf{t}(y) & \text{otherwise} \end{cases} \qquad \kappa'(y) = \begin{cases} (f \circ \kappa)(y) & \text{if } y \succeq x \\ \kappa(y) & \text{otherwise} \end{cases}$$

Then, for every position $y \in \mathsf{pos}(\mathbf{t})$*, d-profile*$(\mathbf{t}|_y)$ $=$ *d-profile*$(\mathbf{t}'|_y)$*, and* κ' *is a correct certificate for* $\mathbf{t}'$*.*

The following trivial lemma will be useful in the sequel.

LEMMA 4.16. *Given a data tree* $\mathbf{t}$*,* $I, I' \in Inters$*, and* $x, y \in \mathsf{pos}(\mathbf{t})$*,* $x \preceq y$*, such that for every* $(\mathcal{A}, q) \in I$ *there is* $(\mathcal{A}, q') \in I'$ *such that* $q \xrightarrow[x,y]{\mathcal{A}} q'$*, we have that* $[\![I']\!]^{\mathbf{t}|_{y\cdot i}} \subseteq [\![I]\!]^{\mathbf{t}|_x}$ *for every* i*.*

The proof of this Lemma can be found in the online appendix.

PROOF OF PROPOSITION 4.15. First, suppose that y is such that

(1) $y \succeq x$, or

(2) $x \not\succeq y \not\succeq x$.

Then, $\kappa'|_y$ is a correct certificate for $\mathbf{t}'|_y$. This is because in the case (2), $\mathbf{t}|_y = \mathbf{t}'|_y$ and $\kappa|_y = \kappa'|_y$, and in the case (1), $\mathbf{t}'|_y$ is the result of applying the data bijection f to $\mathbf{t}|_y$, and $\kappa'|_y = f \circ \kappa|_y$.

Suppose then that $y \prec x$. We show that for any $k \leq \mathsf{V}$,

(a) if $|[\![I]\!]^{\mathbf{t}|_y}| \geq k$, then there are k distinct data values $d_1, \ldots, d_k \in [\![I]\!]^{\mathbf{t}|_y} \cap [\![I]\!]^{\mathbf{t}'|_y}$, and

(b) if $|[\![I]\!]^{\mathbf{t}'|_y}| \geq k$, then there are k distinct data values $d_1, \ldots, d_k \in [\![I]\!]^{\mathbf{t}|_y} \cap [\![I]\!]^{\mathbf{t}'|_y}$.

Note that (a) and (b) imply that the profiles of y are the same in $\mathbf{t}$ and $\mathbf{t}'$.

For (a), suppose $|[\![I]\!]^{\mathbf{t}|_y}| \geq k$. By correctness of κ there are k distinct data values $d_1, \ldots, d_k \in [\![I]\!]^{\mathbf{t}|_y}$ and k distinct positions $y\cdot i_1, \ldots, y\cdot i_k \in \mathsf{pos}(\mathbf{t})$ such that there are k downward paths starting at $y\cdot i_1, \ldots, y\cdot i_k$ with respective certificates $d_1, \ldots, d_k$. By the choice of d_x, if $d_x \in \{d_1, \ldots, d_k\}$, it cannot be that any witness position of $d_x \in [\![I]\!]^{\mathbf{t}|_y}$ lies within $\mathbf{t}|_x$, as it would mean that $d_x \in \hat{\kappa}(x)$. Thus, $d_1, \ldots, d_k \in [\![I]\!]^{\mathbf{t}'|_y}$.

For (b), assume there are k distinct data values $d_1, \ldots, d_k \in [\![I]\!]^{\mathbf{t}'|_y}$. If none of these is d'_x, then the certificate κ (which is the same as κ' for these values) shows the paths to witness these data values in $\mathbf{t}|_y$ and then $d_1, \ldots, d_k \in [\![I]\!]^{\mathbf{t}|_y}$. If $d'_x \in [\![I]\!]^{\mathbf{t}'|_y}$, as d'_x only appears in $\mathbf{t}|_x$, there must be an intersection I' such that $d'_x \in [\![I']\!]^{\mathbf{t}'|_x}$, and for every $(\mathcal{A}, q) \in I$ there is $(\mathcal{A}, q') \in I'$ with $q \xrightarrow[y,x']{\mathcal{A}} q'$, for $x = x'{\cdot}i$. Hence, $f^{-1}(d'_x) = d_x \in [\![I']\!]^{\mathbf{t}|_x}$. Since $d_x \notin \hat{\kappa}(x)$ by hypothesis, this means (by correctness of κ) that d_x is not in any 'small' intersection, and thus $|[\![I']\!]^{\mathbf{t}|_x}| > \mathsf{V}$, which, by Lemma 4.16, implies that $|[\![I]\!]^{\mathbf{t}|_y}| > \mathsf{V}$. Since for all other data values $d'_x \neq d \in [\![I]\!]^{\mathbf{t}'_y}$ we have that $d \in [\![I]\!]^{\mathbf{t}|_y}$, there must be k data values $d_1, \dots, d_k \in [\![I]\!]^{\mathbf{t}|_y} \cap [\![I]\!]^{\mathbf{t}'|_y}$.

It follows that the certificate κ' is also *correct* for all positions y of $\mathbf{t}'$, since all the intersections I with $|[\![I]\!]^{\mathbf{t}|_y}| \leq \mathsf{V}$ coincide in $\mathbf{t}$ and $\mathbf{t}'$. $\square$

By applying repeatedly Proposition 4.15 we obtain the following result.

Corollary 4.17. *For any data tree $\mathbf{t} = \mathbf{b} \otimes \mathbf{d}$ with correct certificate κ, there exists another data tree $\mathbf{t}' = \mathbf{b} \otimes \mathbf{d}'$ with correct certificate κ' such that for every position $x \in \mathsf{pos}(\mathbf{t})$, d-profile$(\mathbf{t}|_x)$ = d-profile$(\mathbf{t}'|_x)$ and $\mathbf{t}'$ has the disjoint values property.*

Putting together Proposition 4.12 with Corollary 4.17 we hence verify Theorem 4.8 which was the objective of the current Section 4.2.

Now we have all the main ingredients to state the horizontal pumping argument. In the next section we show that every $\mathscr{E}$-DD automaton accepts a data tree whose width is bounded by a fixed function on the size of the automaton, or it does not accept any data tree at all.

4.3 Horizontal pumping

We first bound, for any x, the size of the set of children $\mathcal{C}_x$ necessary for any correct certificate.

Lemma 4.18. *For every correct certificate on a data tree $\mathbf{t}$ and every position $x \in \mathsf{pos}(\mathbf{t})$ there exists a subset $\mathcal{C}_x$ of children of x that is valid and inductive with respect to the certificate, and $|\mathcal{C}_x| \leq \mathsf{L}$, with $\mathsf{L} = (\mathsf{K} \cdot \mathsf{N})^{\mathsf{R}} \cdot p(\mathsf{V}, \mathsf{R}, \mathsf{K}, \mathsf{N})$ for some fixed polynomial p.*

The proof of this Lemma can be found in the online appendix.

Lemma 4.19 (Horizontal pumping). *Let*

—$\mathbf{t}$ *be a data tree accepted by a DD automaton $(\mathcal{R}, \mathcal{V})$,*
—(τ, ρ, ρ_h) *be a detailed $\mathcal{R}$-run and κ a correct certificate,*
—$x \in \mathsf{pos}(\mathbf{t})$ *and $\mathcal{C}_x \subseteq \{x{\cdot}1, \dots, x{\cdot}\#children(\mathbf{t}, x)\}$ be a valid and inductive subset of children positions of x,*
—$x{\cdot}i, x{\cdot}(i+1), \dots, x{\cdot}(i+j)$ *be a set of consecutive siblings with the following properties:*
  —$\rho_h(x{\cdot}i) = \rho_h(x{\cdot}j)$,
  —*none of the positions $x{\cdot}(i+1), \dots, x{\cdot}(i+j)$ is in $\mathcal{C}_x$.*

Then the data tree $\mathbf{t}'$ resulting from the deletion of the subtrees $\mathbf{t}|_{x\cdot(i+1)}, \dots, \mathbf{t}|_{x\cdot(i+j)}$

$$\mathbf{t}' = \mathbf{t} \circ f, \qquad f(y) = \begin{cases} y & \text{if } y \in \mathsf{pos}(\mathbf{t}) \text{ and } \forall t : y \not\succeq x{\cdot}(i+t) \\ x{\cdot}(i+t){\cdot}z & \text{if } x{\cdot}(i+j+t){\cdot}z \in \mathsf{pos}(\mathbf{t}) \end{cases}$$

is also accepted, with the detailed run $(\tau \circ f, \rho \circ f, \rho_h \circ f)$ and correct certificate $\kappa \circ f$ resulting by the deletion of the positions $x{\cdot}(i+1), \dots, x{\cdot}(i+j)$.

Proof. Let $\mathbf{t} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$ and $l = \#\mathit{children}(\mathbf{t}, x)$. The run

$$\rho_h(x{\cdot}1) \cdots \rho_h(x{\cdot}i)\rho_h(i + j + 1) \cdots \rho_h(l)$$

continues to be an accepting run for the NFA $\mathcal{B}_x$ corresponding to $\tau(x)$ on the string $\rho(x{\cdot}1) \cdots \rho(x{\cdot}i)\rho(i{+}j{+}1) \cdots \rho(l)$, and hence $\tau(x)$ correctly labels the position with a valid transition. On the other hand, κ continues to be correct since all the necessary elements of $\mathcal{C}_x$ are preserved after the pruning. $\square$

Corollary 4.20. *If there exists a data tree recognized by a DD $(\mathcal{R}, \mathcal{V})$ with a correct certificate, then there also exists a tree with bounded width which is also recognized by $(\mathcal{R}, \mathcal{V})$ with a correct certificate. The bound is $(\mathsf{K} \cdot \mathsf{N})^{\mathsf{R}} \cdot p(\mathsf{V}, \mathsf{R}, \mathsf{K}, \mathsf{N}, |\tilde{Q}|)$ for some polynomial p.*

The proof of this corollary can be found in the online appendix.

Remark 4.21. The bound of Corollary 4.20 also holds for trees with the disjoint values property, since this property is preserved when a subtree is removed.

We just showed how we can bound the width of a tree with a correct certificate. Unfortunately, bounding the height of the tree is not as simple. Here we will need to make use of the properties showed in Section 4.2.

If a run is such that it can be decorated with a correct certificate that has the disjoint values property, we can show that the acceptance or not of the tree can be decided by inspecting only some *local* conditions between every inner node and its children. This will be the object of the next section, where we will prove that these local properties can be tested in 2ExpTime.

4.4 The emptiness algorithm

In this section we show how to label each node of the tree with some finite information (that we call *tree configuration*). We do this in such a way that testing whether a data tree is accepted or not by a DD automaton amounts to verifying a local property between a node's configuration and the configurations of its children, for every node of the tree. The configuration depends solely on the certificate of the root and its children, and on the state of the run of the transducer. There is a doubly exponential number in the size of the automaton of such configurations. This fact, together with the bounded width of the tree we showed in Corollary 4.20, leads to a decision procedure to test for emptiness. The algorithm runs in 2ExpTime considering R as a parameter, or ExpTime if R is taken as a constant.

By Theorem 4.8, we assume for the rest of this section that we are always working with a data tree $\mathbf{t}$ equipped with: an accepting run ρ of $\mathcal{R}$ on $\mathbf{t}$, and a correct certificate κ on $\mathbf{t}$, under the disjoint values property.

In the next section we define the configurations that we associate to each node. We will show an algorithm to test if there is a tree with an accepting configuration at the root. This algorithm heavily relies on the fact that the tree is ranked, given by Corollary 4.20. The correctness of this algorithm will be a consequence of the disjoint values property and the admisibility of correct certificates shown in Section 4.2.

Configurations

Next, we define a configuration for a data tree. The idea is that each position x of a data tree $\mathbf{t}$ is associated with the configuration for $\mathbf{t}|_x$. Let $\mathbf{t} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$. We describe what a configuration looks like for a A **tree configuration** contains the state of the transducer's run, the data value of the root, the root's certificate, the children's certificates, and the description of all the data values of the certificates it contains. In the following definition remember that $\hat{\kappa}(x)$ denotes the set consisting of $\kappa(x)$ and all $\kappa(x')$ for x' a children of x, and that $\rightharpoonup$ defines partial functions.

$$\mathit{TConfigs} = \dot{Q} \times \mathbb{D} \times \mathbb{D} \times \wp_{\leq \mathsf{W}}(\mathbb{D}) \times (\mathbb{D} \rightharpoonup \wp(\mathit{Aut} \times \mathcal{Q}))$$
$$\mathit{tconfig}(\mathbf{t}, \rho, \kappa) = \big(\, \rho(\epsilon),\, \mathbf{d}(\epsilon),\, \kappa(\epsilon),\, \hat{\kappa}(\epsilon),\, \{d \mapsto \mathrm{desc}_{\mathbf{t}}(d) \mid d \in \hat{\kappa}(\epsilon)\}\, \big)$$

Although the configurations contain data values, it is not important to know the concrete data values. We are only interested in the classes of equivalence modulo equality contained in the configuration. This is sensible, since the model of automata presented can only test for data equality or inequality. Later on, we will see that this means that we can substitute $\mathbb{D}$ with a *finite* alphabet.

The objective is to prove that if we are given a tree with a run and certificate, we can deduce the configuration of the root by inspecting only the configurations of the immediate subtrees. And vice versa, if we are given a forest of trees with their respective runs and configurations, and a configuration that is compatible with them (in a sense that will be described below), we can then build a witness data tree with the configuration in question.

Given a *DD* automaton $(\mathcal{R}, \mathcal{V})$, what conditions on the configurations do we need to check? To abstract these conditions we define an entailment relation that checks whether the root configuration can be deduced from the configurations of the children

$$\vdash \;\subseteq\; (\mathit{TConfigs})^* \times \mathit{TConfigs}\ .$$

We next give the conditions for $M \vdash (\dot{q}_0, d_0, c_0, C_0, \alpha_0)$ to hold, with $(\dot{q}_0, d_0, c_0, C_0, \alpha_0)$ the configuration of the root and $M = (\dot{q}_1, d_1, c_1, C_1, \alpha_1) \cdots (\dot{q}_m, d_m, c_m, C_m, \alpha_m)$ the configurations of the immediate subtrees.

We define that $M \vdash (\dot{q}_0, d_0, c_0, C_0, \alpha_0)$ holds if the following conditions are satisfied. The conditions, although lengthy, are straightforward. They are necessary and sufficient to have a tree where $(\dot{q}_0, d_0, c_0, C_0, \alpha_0)$ is the configuration of the root, and M are the configurations of the children of the root. They can be informally described as follows: (i) $\vdash$ is consistent with the run of the transducer; (ii) c_0 is a fresh data value (*i.e.*, a data value that is not in M), or a data value equal to some c_i; (iii) every data value c_i is contained in C_0; (iv) the c_i's have the *validity* property for every intersection; (v) c_0 and the c_i's have the *inductivity* property; (vi) α_0 is obtained from the description at the children configuration α_i's, by applying all possible transitions from any of the automata; (vii) the root satisfies the verifier's formula. In the next definition, we use a function

$$f : (\mathit{TConfigs})^* \to (\mathbb{B} \times \mathbb{D}) \to \mathbb{D} \to \wp(\mathit{Aut} \times \mathcal{Q})$$

such that $f(M)(b, d_0)(d) = A$ if A is the data description of d, deduced from the root's letter and datum (b, d_0), and the configurations of the children positions M.

$$\begin{aligned} f(M)(b,d_0)(d) = \{(\mathcal{A}, q) \mid i \in [m], d \in C_i, (\mathcal{A}, q') \in \alpha_i(d), (q,b,q') \in \mathcal{A}\} \cup \\ \{(\mathcal{A}, q) \mid d = d_0, (q,b,q') \in \mathcal{A}, q' \in \mathcal{Q}_F^{\mathcal{A}}\} \end{aligned}$$

We also use

$$\hbar : (\mathit{TConfigs})^* \to (\mathbb{B}\times\mathbb{D}) \to \mathit{Inters} \to \wp(\mathbb{D})$$

where $\hbar(M)(b,d_0)(d)$ is the set of all data values in M that satisfy the intersection I at the root.

$$\hbar(M)(b,d_0)(I) = \{d \mid d \in \{d_0\} \cup C_1 \cup \cdots \cup C_m,\ I \subseteq f(M)(b,d_0)(d)\}$$

The conditions are as follows.

(i) There exists $(\dot{q}_0, a, b, \mathcal{L}) \in \delta$ with $\dot{q}_1 \cdots \dot{q}_m \in \mathcal{L}$. For the remaining conditions let us fix $f' = f(M)(b,d_0)$ and $\hbar' = \hbar(M)(b,d_0)$.

(ii) Either $c_0 = c_i$ for some $i \in [m]$, or $c_0 = d_0$ otherwise.

(iii) $C_0 = \{c_0, c_1, \ldots, c_m\}$.

(iv) For every $I \in \mathit{Inters}$, the validity condition (Definition 4.5) holds,

$$\mathit{valid}(\hbar'(I), I, (b, d_0, c_0), \cup_{i\in[m]}\{(c_i, \alpha_i(c_i))\}) \ .$$

(v) The inductive condition (Definition 4.6) holds,

$$\mathit{inductive}(\alpha_0(c_0), (b, d_0, c_0), \cup_{i\in[m]}\{(c_i, \alpha_i(c_i))\}) \ .$$

(vi) $\alpha_0 = \{d \mapsto f'(d) \mid d \in C_0\}$.

(vii) $\{I \mapsto |\hbar'(I)|_{\leq \mathsf{V}}\} \models \mathsf{v}(b)$, where v is the verifier's mapping, and $\models$ is as defined in Table I on page 15.

We remark that in the above definition we do not exclude the case where $M = \epsilon$. In fact, this case corresponds to the configurations of the leaves.

Correctness of $\vdash$

We verify that this is indeed enough to have a decision procedure. Below, the *soundness* Proposition 4.22 states that, given a sequence of trees with their respective configurations, for any configuration entailed from these we can find a tree that witnesses this configuration. On the other hand, the *completeness* Proposition 4.25 states that for any tree, the configurations of the immediate subtrees entail the configuration of the tree.

PROPOSITION 4.22 (SOUNDNESS). *Given m data trees with $\mathcal{R}$-runs, correct certificates and the disjoint values property*

$$\mathbf{t}_1, \rho_1, \kappa_1, \ldots, \mathbf{t}_m, \rho_m, \kappa_m$$

with $\mathbf{t}_i = \mathbf{a}_i \otimes \mathbf{b}_i \otimes \mathbf{d}_i \in \mathit{Trees}(\mathbb{A}\times\mathbb{B}\times\mathbb{D})$ such that ρ_i is a run for $\mathbf{a}_i \otimes \mathbf{b}_i$, $\mathcal{V}$ accepts $\mathbf{b}_i \otimes \mathbf{d}_i$, and

$$\forall i \neq j \quad \mathit{data}(\mathbf{t}_i) \cap \mathit{data}(\mathbf{t}_j) \ \subseteq\ \hat{\kappa}_i(\epsilon) \cap \hat{\kappa}_j(\epsilon) \ . \qquad (2)$$

If

$$M = (\dot{q}_1, d_1, c_1, C_1, \alpha_1) \cdots (\dot{q}_m, d_m, c_m, C_m, \alpha_m) \vdash T$$

with $(\dot{q}_i, d_i, c_i, C_i, \alpha_i) = \mathit{tconfig}(\mathbf{t}_i, \rho_i, \kappa_i)$ *for every* i*, then there exists a tree* $\mathbf{t}_0 \in \mathit{Trees}(\mathbb{A}{\times}\mathbb{B}{\times}\mathbb{D})$ *with run* ρ_0 *and correct certificate* κ_0 *with the disjoint values property, such that it is accepted by* $\mathcal{V}$*, and*

$$\mathit{tconfig}(\mathbf{t}_0, \rho_0, \kappa_0) = T.$$

Before going into the details of the proof, we need to state some necessary lemmas (4.23, 4.24). Take any tree $\mathbf{t}_0$ with root (a, b, d_0) and subtrees $\mathbf{t}_1, \dots, \mathbf{t}_m$, such that

$$\text{either} \quad d_0 \in C_1 \cup \cdots \cup C_m \quad \text{or} \quad d_0 \notin \cup_{i\in[m]} \mathit{data}(\mathbf{t}_i)\ . \tag{3}$$

Let us fix $f' = f(M)(b, d_0)$ and $\hbar' = \hbar(M)(b, d_0)$.

The proof of the following two Lemmas can be found in the online appendix.

Lemma 4.23. *For any* $d \in C_1 \cup \cdots \cup C_m$*,* $f'(d) = \mathit{desc}_{\mathbf{t}_0}(d)$*.*

Lemma 4.24. *For any intersection* I *and* $k \leq \mathsf{V}$*,* $|[\![I]\!]^{\mathbf{t}_0}| \geq k$ *iff* $|\hbar'(I)| \geq k$*.*

Proof of Proposition 4.22. We show that there exists $(a, b, d_{cert}) \in \mathbb{A}{\times}\mathbb{B}{\times}\mathbb{D}$ such that the tree with (a, b, d_{cert}) at the root and $\mathbf{t}_1, \dots, \mathbf{t}_m$ as immediate subtrees has a correct certificate κ_0 with the disjoint values property, and a $\mathcal{R}$-run ρ_0, such that the configuration of the tree is T.

We first define the run. Let $T = (\dot{q}_0, d_0, d_{cert}, C_0, \alpha_0)$. As we remarked before, we are only interested in the classes of equivalence of $T, T_1, \dots, T_m$ modulo renaming of data values. So that we can always assume that d_0 is such that hypothesis (3) holds: if $d_0 \notin C_1 \cup \cdots \cup C_m$, we simply assume that d_0 is any value not contained in $\mathit{data}(\mathbf{t}_1) \cup \cdots \cup \mathit{data}(\mathbf{t}_m)$.

By condition (i) of the entailment definition, there is some transition $(\dot{q}_0, a, b, \mathcal{L})$ of $\mathcal{R}$ such that $\rho_1(\epsilon) \cdots \rho_m(\epsilon) \in \mathcal{L}$. Let $\mathbf{t}_0 \in \mathit{Trees}(\mathbb{A}{\times}\mathbb{B}{\times}\mathbb{D})$, defined by $\mathbf{t}_0(\epsilon) = (a, b, d_0)$ and $\mathbf{t}_0(i{\cdot}x) = \mathbf{t}_i(x)$ for $i \in [m]$. Hence the run ρ_0 defined as $\rho_0(\epsilon) = \dot{q}_0$, $\rho_0(i{\cdot}x) = \rho_i(x)$ is a valid $\mathcal{R}$-run on $\mathbf{t}_0$.

We now show that T is indeed a configuration that corresponds to $\mathbf{t}_0$. By condition (ii) either $d_{cert} = d_0$ for some fresh data value (*i.e.*, not appearing in any subtree, by hypothesis (3)), or $d_{cert} = \kappa_i(\epsilon)$ for some $i \in [m]$. In the first case we trivially have

$$\mathrm{desc}_{\mathbf{t}_0}(d) = \{(\mathcal{A}, q) \mid (q, b, q') \in \mathcal{A}, q' \in Q_F^{\mathcal{A}}\} = \alpha_0(d)$$

by definition of α_0 (condition (vi)) and definition of $f'(d_{cert})$. In the second case, we have by Lemma 4.23 that $f'(d_{cert})$ correctly yields the description of d_{cert} at the root and we verify $\mathrm{desc}_{\mathbf{t}_0}(d_{cert}) = \alpha_0(d_{cert})$. By condition (iii) and using the same reasoning as before, $\kappa_1(\epsilon) \cup \cdots \cup \kappa_m(\epsilon) = C_0$ and for all $d \in C_0$, $\alpha_0(d) = \mathrm{desc}_{\mathbf{t}_0}(d)$. Then, we have that $\mathit{dconfig}(\mathbf{t}_0, \rho_0, \kappa_0) = T$.

We define the certificate κ_0 as $\kappa_0(\epsilon) = d_{cert}$ and $\kappa_0(i{\cdot}x) = \kappa_i(x)$. From hypothesis (2) and the fact that $\mathbf{t}_i, \kappa_i$ have the disjoint values property for every $i \in [m]$, we deduce that $\mathbf{t}_0, \kappa_0$ also has the disjoint values property.

We show that κ_0 is a *correct* certificate for $\mathbf{t}_0$. For the *validity* condition we first have by Lemma 4.24 that the sets $\hbar'(I)$ are good approximations[6] of $[\![I]\!]^{\mathbf{t}_0}$. This, together with condition (iv) and Definition 4.5 gives us that κ_0 must be *valid*, while the *inductivity* property is a consequence of Definition 4.6 and condition (v).

[6]They coincide in $\min([\![I]\!]^{\mathbf{t}_0}, \mathsf{V})$ elements.

Finally, since $\hbar'(I)$ has the same cardinality up to V elements as $[\![I]\!]^{\mathbf{t}_0}$, $\{I \mapsto |\hbar'|_{\leq \mathsf{V}}\} \models \varphi$ iff φ holds at $\mathbf{t}_0$ for any $\varphi \in \Phi$ used in $\mathcal{V}$. Then, by condition (vii), $\mathbf{t}_0$ is accepted by $\mathcal{V}$. □

Proposition 4.25 (completeness). *Given a data tree $\mathbf{t} \in Trees(\mathbb{A}\times\mathbb{B}\times\mathbb{D})$ with correct certificate κ, run ρ and the disjoint values property accepted by $\mathcal{V}$, then*

$$tconfig(\mathbf{t}|_1, \rho|_1, \kappa|_1) \cdots tconfig(\mathbf{t}|_m, \rho|_m, \kappa|_m) \vdash tconfig(\mathbf{t}, \rho, \kappa)$$

for m the maximum index such that $m \in \mathsf{pos}(\mathbf{t})$.

Proof. Let $\mathbf{t} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$. We verify conditions (i) through (vii). Condition (i) is trivially true as ρ is a run on $\mathbf{t}$. Condition (ii) holds, since the $\kappa(\epsilon)$ can be either equal to $\mathbf{d}(\epsilon)$ or equal to some child certificate $\kappa(i)$ as a consequence of κ being a correct certificate. Condition (iii) holds because we have all the certificates from the child configurations. The correctness of the descriptions of condition (vi) for the data values $\{\kappa(1), \ldots, \kappa(m)\}$ is based on the disjoint values property. As a consequence of this property, we have that for every data value $\kappa(i)$ and every $j \in [m]$, if $\kappa(i) \in data(\mathbf{t}|_j)$, then $\kappa(i) \in \hat{\kappa}(j)$. This means that we have a complete description of $\kappa(i)$ for every subtree $\mathbf{t}|_j$. For the case of the root's certificate $\kappa(\epsilon)$ we have two cases. If $\kappa(\epsilon)$ equals some $\kappa(i)$, then we use the same argument as before. Otherwise, we use the *inductivity* of the certificate κ, knowing that by Definition 4.6 if $\kappa(\epsilon)$ is in some $[\![\mathcal{A}, q]\!]^{\mathbf{t}}$, there must be a path of certificates with value $\kappa(\epsilon)$. So, the fact that there are no $\kappa(i) = \kappa(\epsilon)$ means that $\mathrm{desc}_{\mathbf{t}}(\kappa(\epsilon))$ can be completely witnessed locally by inspecting only the root. Then, the description obtained by f' contained in $tconfig(\mathbf{t}, \rho, \kappa)$ is correct. Since condition (vi) holds and $\mathbf{b} \otimes \mathbf{d}$ satisfies $\mathsf{v}(\mathbf{b}(\epsilon))$, then it verifies condition (vii). Finally, conditions (iv) and (v) are consequences of the validity and inductivity properties of κ respectively. □

Remark 4.26. Observe that Corollary 4.20 with Remark 4.21 gives us a bound on the width of a recognized tree with a correct run and the disjoint values property. We can thus restrict ourselves to relations $T_1 \cdots T_t \vdash T$ with $t \leq \mathsf{W}$ from now on. (Remember that W is the bound on the width of the tree given by Corollary 4.20.)

Also, note that the concrete data values of the configuration are not important and can be abstracted away, as soon as they allow to test the conditions of the entailment $\vdash$. For $T, T' \in TConfigs$, let us write $T \sim T'$ if there is a bijection of data values $f : \mathbb{D} \to \mathbb{D}$ such that $f(T) = T'$, where $f(T)$ stands for the replacement of every datum d by $f(d)$ in T. In every configuration there are at most $\mathsf{W}+2$ data values (the root's data value, the root's certificate, and at most W corresponding to the certificates of the children). Then, by Remark 4.26, a '$\vdash$' test involves not more than $\mathsf{W}+1$ configurations and hence we only need at most $(\mathsf{W}+1)\cdot(\mathsf{W}+2)$ different data values. Let us define $TConfigs'$ to be $TConfigs$ where instead of having $\mathbb{D}$ as data domain, we have $\{1, \ldots, (\mathsf{W}+1)\cdot(\mathsf{W}+2)\}$. Let

$$\mathbb{D}' := \{1, \ldots, (\mathsf{W}+1)\cdot(\mathsf{W}+2)\} \tag{4}$$

and then let $TConfigs'$ be defined in terms of the restricted set of data values $\mathbb{D}'$,

$$TConfigs' = \dot{Q} \times \mathbb{D}' \times \mathbb{D}' \times \wp_{\leq \mathsf{W}}(\mathbb{D}') \times (\mathbb{D}' \rightharpoonup \wp(Aut \times \mathcal{Q})) \ . \tag{5}$$

We then have the following obvious lemma.

LEMMA 4.27. *For every $T, T_1, \ldots, T_n \in \mathit{TConfigs}$ such that $T_1 \cdots T_n \vdash T$, there exist $T', T'_1, \ldots, T'_n \in \mathit{TConfigs}'$ with $T \sim T'$ and $T_i \sim T'_i$ for all i, such that $T'_1 \cdots T'_n \vdash T'$.*

This means that, since we are only interested in the tree configurations that can be reached by $\vdash$ modulo isomorphism of data values, we can simply use the tree configurations of $\mathit{TConfigs}'$. These are doubly exponential in R, or singly exponential if R is fixed.

LEMMA 4.28. *The number of elements in $\mathit{TConfigs}'$ is exponential in $|\mathcal{R}|$ and $|\mathcal{V}|$ if R is a constant, or doubly exponential otherwise.*

The proof of this Lemma can be found in the online appendix.

As the first step towards an upper bound, we observe that the $\vdash$ relation on $\mathit{TConfigs}'$ can be checked in polynomial time in $\mathsf{Aut}, \mathsf{W}, |\mathcal{R}|$.

LEMMA 4.29. *Given $T, T_1, \ldots, T_n \in \mathit{TConfigs}'$ with $n \leq \mathsf{W}$, $T_1 \cdots T_n \vdash T$ can be tested in time $p(\mathsf{Aut}, \mathsf{W}, |\mathcal{R}|)$ for some polynomial p.*

We now show an algorithm to test whether a tree configuration can be reached by the entailment relation $\vdash$.

THEOREM 4.30. *The emptiness problem for DD automata is in* 2EXPTIME. *It can be tested in time*

$$(\mathsf{Aut}, |\mathcal{R}| \cdot \mathsf{V} \cdot \mathsf{R} \cdot \mathsf{K} \cdot \mathsf{N})^{p(|\tilde{Q}|, \mathsf{V}, \mathsf{R}) \cdot r(\mathsf{K} \cdot \mathsf{N})^{s(\mathsf{R})}}$$

for p, r and s polynomials.

PROOF. We consider a standard reachability algorithm by saturation. We start with an initial empty set of configurations $C_0 = \emptyset$, and we iterate to make it grow to entailed configurations until, after at most $|\mathit{TConfigs}'|$ iterations, the set stabilizes. We then test if some of the reachable tree configurations contains a final state.

The set of initial configurations is $C_0 = \emptyset$. At iteration $i + 1$, for every possible $T_0 \in \mathit{TConfigs}'$ we test the following conditions

—$T_0 \notin C_i$

—There exists a (possibly empty) sequence $T_1, \ldots, T_t$ with $t \leq \mathsf{W}$ such that
  —$T_1, \ldots, T_t \in C_i$
  —$T_1, \ldots, T_t \vdash T_0$

and we define $C_{i+1} := C_i \cup C'$, for C' the set of all configurations T_0 satisfying the above conditions. If C' is empty, we stop and return the subset C_i of $\mathit{TConfigs}'$ of $(\vdash)$-reachable configurations.

This algorithm clearly gives as a result the set of configurations of all the accepted trees. For every iteration we might need to perform $|\mathit{TConfigs}'|^{\mathsf{W}+1}$ tests for the $\vdash$ conditions, each one demanding $p(\mathsf{Aut}, \mathsf{W}, |\mathcal{R}|)$ for a polynomial p by Lemma 4.29. Finally, the loop can only be executed $|\mathit{TConfigs}'|$ times. We then have that the total time consumed is

$$|\mathit{TConfigs}'|^{\mathsf{W}+2} \cdot p(\mathsf{Aut}, \mathsf{W}, |\mathcal{R}|) \ .$$

By the inequation (7) of Lemma 4.28, and since W is exponential only in R by (6), we have that the emptiness problem is bounded by

$$(\mathsf{Aut} \cdot |\mathcal{R}| \cdot \mathsf{V} \cdot \mathsf{R} \cdot \mathsf{K} \cdot \mathsf{N})^{p(|\tilde{Q}|,\mathsf{V},\mathsf{R}) \cdot r(\mathsf{K}\cdot\mathsf{N})^{s(\mathsf{R})}}$$

for p, r, s polynomials, and we hence have a 2ExpTime decision procedure. We just proved that the problem of whether a DD automaton accepts a tree $\mathbf{t}$ with a correct certificate with the disjoint values properties, can be tested in 2ExpTime. Then, by Theorem 4.8 the result follows. □

Note that the theorem above implies the Main Theorem 4.1. From the previous proof, we have that the complexity is doubly exponential only in R.

Corollary 4.31. *If* R *is fixed, emptiness of DD automata is in* ExpTime. *It can be tested in time bounded by*

$$(\mathsf{Aut} \cdot |\mathcal{R}| \cdot \mathsf{V} \cdot \mathsf{K} \cdot \mathsf{N})^{p(|\tilde{Q}|,\mathsf{V},\mathsf{K},\mathsf{N})}$$

for some polynomial p.

Note that the height of a $\vdash$ derivation is directly related to the height of the tree. Hence, for trees with a fixed height, we can take advantage of this fact by performing an on-the-fly algorithm.

Definition 4.32. We define the **height-h emptiness problem** as follows.

	Height-h emptiness problem
Input:	A DD automaton $(\mathcal{R}, \mathcal{V})$ and a number $h \in \mathbb{N}$.
Output:	Is there a data tree $\mathbf{t}$ of height at most h such that $\mathbf{t}$ is accepted by $(\mathcal{R}, \mathcal{V})$?

For the next theorem let us assume that h is coded in unary.

Theorem 4.33. *If* R *is fixed, the height-h emptiness problem of DD automata is in* PSpace.

The proof of this Theorem can be found in the online appendix.

Remark 4.34. If $sp(\vdash)$ is the space needed to check $T_1 \cdots T_n \vdash T$, $n \leq \mathsf{W}$, then the algorithm of Theorem 4.33 uses an amount of space bounded by

$$sp(\vdash) + p(h, \log(|\dot{Q}|), |\tilde{Q}|, \mathsf{K}, \mathsf{N}, \mathsf{V})$$

for some polynomial p.

The purpose of the remark above for discriminating the space needed to perform the entailment condition $sp(\vdash)$ will become clear in Section 6.2.

5. XPATH

Here we introduce the query language XPath adapted to data trees, though originally it is a language for XML documents. In Section 6.3 we will see that the satisfiability problem on data trees is equivalent to the satisfiability problem on XML documents. We work with a simplification of XPath, stripped of its syntactic sugar. We consider fragments of XPath that correspond to the navigational part of

$$
\begin{array}{rlrl}
[\![\downarrow]\!]^{\mathbf{t}} &= \{(x, x{\cdot}i) \mid x{\cdot}i \in \mathsf{pos}(\mathbf{t})\} & [\![\alpha^*]\!]^{\mathbf{t}} &= \text{the reflexive transitive closure of } [\![\alpha]\!]^{\mathbf{t}} \\
[\![\varepsilon]\!]^{\mathbf{t}} &= \{(x, x) \mid x \in \mathsf{pos}(\mathbf{t})\} & [\![\alpha\beta]\!]^{\mathbf{t}} &= \{(x, z) \mid \exists y. \\
[\![\alpha \cup \beta]\!]^{\mathbf{t}} &= [\![\alpha]\!]^{\mathbf{t}} \cup [\![\beta]\!]^{\mathbf{t}} & & \quad (x, y) \in [\![\alpha]\!]^{\mathbf{t}}, (y, z) \in [\![\beta]\!]^{\mathbf{t}}\} \\
[\![\alpha[\varphi]]\!]^{\mathbf{t}} &= \{(x, y) \in [\![\alpha]\!]^{\mathbf{t}} \mid y \in [\![\varphi]\!]^{\mathbf{t}}\} & [\![[\varphi]\alpha]\!]^{\mathbf{t}} &= \{(x, y) \in [\![\alpha]\!]^{\mathbf{t}} \mid x \in [\![\varphi]\!]^{\mathbf{t}}\} \\
[\![a]\!]^{\mathbf{t}} &= \{x \in \mathsf{pos}(\mathbf{t}) \mid \mathbf{a}(x) = a\} & [\![\langle\alpha\rangle]\!]^{\mathbf{t}} &= \{x \in \mathsf{pos}(\mathbf{t}) \mid \exists y.(x, y) \in [\![\alpha]\!]^{\mathbf{t}}\} \\
[\![\neg\varphi]\!]^{\mathbf{t}} &= \mathsf{pos}(\mathbf{t}) \setminus [\![\varphi]\!]^{\mathbf{t}} & [\![\varphi \wedge \psi]\!]^{\mathbf{t}} &= [\![\varphi]\!]^{\mathbf{t}} \cap [\![\psi]\!]^{\mathbf{t}} \\
[\![\langle\alpha{=}\beta\rangle]\!]^{\mathbf{t}} &= \{x \in \mathsf{pos}(\mathbf{t}) \mid \exists y, z.(x, y) \in [\![\alpha]\!]^{\mathbf{t}}, & [\![\langle\alpha{\neq}\beta\rangle]\!]^{\mathbf{t}} &= \{x \in \mathsf{pos}(\mathbf{t}) \mid \exists y, z.(x, y) \in [\![\alpha]\!]^{\mathbf{t}}, \\
& \quad (x, z) \in [\![\beta]\!]^{\mathbf{t}}, \mathbf{d}(y) = \mathbf{d}(z)\} & & \quad (x, z) \in [\![\beta]\!]^{\mathbf{t}}, \mathbf{d}(y) \neq \mathbf{d}(z)\}
\end{array}
$$

Table II. Semantics of XPath for a data tree $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$.

XPath 1.0 with data equality and inequality. Let us give the formal definition of this logic. XPath is a two-sorted language, with *path* expressions (that we write α, β, γ) and *node* expressions (φ, ψ, η). The fragment XPath($\mathcal{O}, =$), with $\mathcal{O} \subseteq \{\downarrow, \downarrow_*\}$ is defined by mutual recursion as follows:

$$
\begin{array}{lll}
\alpha, \beta & ::= \; o \mid \alpha[\varphi] \mid [\varphi]\alpha \mid \alpha\beta \mid \alpha \cup \beta & o \in \mathcal{O} \cup \{\varepsilon\}\ , \\
\varphi, \psi & ::= \; a \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \langle\alpha\rangle \mid \langle\alpha = \beta\rangle \mid \langle\alpha \neq \beta\rangle & a \in \mathbb{A},
\end{array}
$$

where $\mathbb{A}$ is a finite alphabet. A *formula* of XPath($\mathcal{O}, =$) is either a node expression or a path expression of the logic. XPath($\mathcal{O}$) is the fragment XPath($\mathcal{O}, =$) without the node expressions of the form $\langle\alpha = \beta\rangle$ or $\langle\alpha \neq \beta\rangle$.

There have been efforts to extend this navigational core of XPath in order to have the full expressivity of FO or MSO —for example by adding a least fix-point operator (*cf.* [ten Cate 2006, Sect. 4.2])— but these logics generally lack clarity and simplicity. However, a form of recursion can be added by means of the Kleene star, which allows the formation of the transitive closure of any path expression. Although in general this is not enough to already capture MSO —as shown by ten Cate and Segoufin [2008], it does give an intuitive language with counting ability. By regXPath($\mathcal{O}, =$) we refer to the language where path expressions are extended

$$\alpha, \beta \; ::= \; o \mid \alpha[\varphi] \mid [\varphi]\alpha \mid \alpha\beta \mid \alpha \cup \beta \mid \alpha^* \quad o \in \mathcal{O}$$

by allowing the Kleene star on *any* path expression. In terms of expressivity, we have that XPath($\downarrow_*, =$) $\subsetneq$ XPath($\downarrow_*, \downarrow, =$) $\subsetneq$ regXPath($\downarrow, =$) $=$ regXPath($\downarrow_*, \downarrow, =$). For example, in regXPath($\downarrow, =$) we can express that the tree has a branch of even length. This is a property that does not depend on the data values, and that cannot be expressed in XPath($\downarrow_*, \downarrow$) nor XPath($\downarrow_*, \downarrow, =$).

We formally define the semantics of XPath in Table II. As an example, if $\mathbf{t}$ is the data tree depicted by Figure 1 on page 5,

$$[\![\langle \downarrow_*[\, b \wedge \langle \downarrow[b] \neq \downarrow[b] \rangle \,] \rangle]\!]^{\mathbf{t}} = \{\varepsilon, 1, 12\}.$$

Hereinafter, we write $\mathbf{t} \models \varphi$ to denote $[\![\varphi]\!]^{\mathbf{t}} \neq \emptyset$. In this case we say that $\mathbf{t}$ *satisfies* φ. We state the problem we will address, given a fragment $\mathscr{P}$ of XPath.

SAT-$\mathscr{P}$	Satisfiability problem for $\mathscr{P}$
INPUT:	$\varphi \in \mathscr{P}$.
OUTPUT:	Is there a tree $\mathbf{t}$ such that $\mathbf{t} \models \varphi$?

As we are working with downward-looking fragments of XPath, this problem is equivalent to testing if there is a tree in which the formula φ is satisfied at the *root*. Moreover, we can restrict ourselves to the case where φ is a *node expression*, since $[\![\alpha]\!] \neq \emptyset$ iff $[\![\langle\alpha\rangle]\!] \neq \emptyset$. We remind the reader that although we state the problem in terms of data trees, all our results hold on the class of all XML documents. Indeed, this is a consequence of considering an XML document as a data tree where the attributes are at leaf positions. This issue will be explained in detail in Section 6.3.

6. THE SATISFIABILITY PROBLEM

The first part of this section is devoted to the proof of decidability of the satisfiability problem for regXPath$(\downarrow, =)$, the language with the child relation and the Kleene star over path expressions. In later subsections we consider the satisfiability problem of several fragments of this logic, with or without data tests.

6.1 Regular-downward XPath

The proof of satisfiability for regXPath$(\downarrow, =)$ is by reduction to the emptiness problem of DD automata. Before embarking on the reduction, we need to fix some standard terminology.

Definition 6.1. A **subformula** of φ is a substring of φ that is a formula. We say that a set S of formulæ is **closed under subformulæ** if: for every $\varphi \in S$ and for every subformula ψ of φ, we have that $\psi \in S$. S is **closed under simple negations** if, for every $\varphi \in S$ it holds $\neg\varphi \in S$ unless φ is of the form $\neg\psi$. We denote the minimal superset of S closed under subformulæ and simple negations by $S^{\neg}$.

A **locally consistent set** over S is a maximal subset of $H \subseteq S$ that satisfies the following conditions:

—For all $\neg\varphi \in S$: $\neg\varphi \in H$ if and only if $\varphi \notin H$.

—For all $\varphi \wedge \psi \in S$: $\varphi \in H$ and $\psi \in H$ if and only if $\varphi \wedge \psi \in H$.

—For all $\varphi \vee \psi \in S$: $\varphi \in H$ or $\psi \in H$ if and only if $\varphi \vee \psi \in H$.

For the rest of the section, we consider the parameters of the DD automata (K, N, R, V, $|\dot{Q}|$, $|\tilde{Q}|$, $|\mathcal{R}|$, Aut, $|\mathcal{V}|$) as defined in Section 4.

Theorem 6.2. *Given a formula $\eta \in$ regXPath$(\downarrow, =)$, a DD automaton $(\mathcal{R}, \mathcal{V})$ can be effectively built, such that for any data tree $\mathbf{t}$: $\mathbf{t} \models \eta$ iff $(\mathcal{R}, \mathcal{V})$ accepts $\mathbf{t}$.*

Proof. Let η be a formula of regXPath$(\downarrow, =)$. We build the DD automaton $(\mathcal{R}, \mathcal{V})$, where $\mathcal{R}$ tags each node with those sub-node expressions of η that hold at each node, and $\mathcal{V}$ checks that all the data and path expressions are verified.

Let $\mathsf{nsub}(\eta) = \{\gamma \mid \gamma$ a node expression in $\mathsf{sub}(\eta)\}$, where $\mathsf{sub}(\eta)$ is the set of subformulæ of η. Let $\mathbb{B}$ be the set of all locally consistent sets over $\{\eta\}^{\neg}$. Let us build $\mathcal{R}$ in such a way that at each step it chooses nondeterministically one element from $\mathbb{B}$ consistent with the current label and outputs it. That is, if $a \in \mathbb{A}$ is the letter of the current position then it outputs any element $b \in \mathbb{B}$ such that $\neg a \notin b$.[7] Note that the transducer only needs one (final) state (*i.e.*, $\dot{Q} = \{\dot{q}\}$) to reflect this

[7]Note that if $a \notin \mathsf{sub}(\eta)$, then $\neg a \notin b$ and $a \notin b$ for all $b \in \mathbb{B}$.

behavior. Further, the only regular language used in all its transitions is $\{\dot{q}\}^*$. We can represent $\{\dot{q}\}^*$ with a NFA with a singleton set of states $\tilde{Q}$.

For every path expression α in $\mathsf{sub}(\eta)$, let $\mathcal{A}_\alpha$ be a NFA over the alphabet $\mathbb{B}$ that recognizes α. For example, if $\alpha = [\varphi] \downarrow [\psi]$, then $\mathcal{A}_\alpha$ recognizes $\{b\,b' \mid b, b' \in \mathbb{B}, \varphi \in b, \psi \in b'\}$. It can be built in polynomial time in $|\alpha|$ and $|\mathbb{B}|$ (but note that $|\mathbb{B}|$ is exponential in $|\eta|$). We define the verifier $\mathcal{V}$ to contain the set $Aut = \{\mathcal{A}_\alpha \mid \alpha$ path expression of $\mathsf{sub}(\eta)\}$ of automata. The mapping v is defined, for every element $b \in \mathbb{B}$, as the formula that tests all the path formulæ in b.

$$\mathsf{v}(b) = \bigwedge_{\substack{(\neg)\langle\alpha\odot\beta\rangle\in b\\ \odot\in\{=,\neq\}}} (\neg)\,\exists v, v' \,.\, \big(v \odot v' \wedge D_\alpha(v) \wedge D_\beta(v')\big) \quad \wedge \quad \bigwedge_{(\neg)\langle\alpha\rangle\in b} (\neg)\,\exists v \,.\, D_\alpha(v)$$

Intuitively, the only purpose of the transducer $\mathcal{R}$ is to *guess* which subformulæ of η are true and which are false. The real work is done by the verifier $\mathcal{V}$, checking that every formula was correctly guessed.

Suppose that $\mathbf{t} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$ is accepted by $(\mathcal{R}, \mathcal{V})$, *i.e.*, $\mathbf{a} \otimes \mathbf{b} \in \mathcal{R}$, $\mathbf{b} \otimes \mathbf{d} \in \mathcal{V}$. For every position $x \in \mathsf{pos}(\mathbf{t})$ and subformula $\psi \in \mathsf{sub}(\eta)$, we show that $\psi \in \mathbf{b}(x)$ if and only if $x \in [\![\psi]\!]^{\mathbf{a}\otimes\mathbf{d}}$. We proceed by induction on the size of ψ. The base case is when ψ is a test for a label. This is immediate by the definition of $\mathcal{R}$, which preserves the label: $\mathbf{a}(x) \in \mathbf{b}(x)$ if $\mathbf{a}(x) \in \mathsf{sub}(\eta)$. Suppose now that $\psi = \langle\alpha \odot \beta\rangle$ with $\odot \in \{=, \neq\}$. By definition of $\mathsf{v}(\mathbf{b}(x))$, $\mathcal{V}$ verifies that there are data values $d \odot d'$ such that $d \in [\![\mathcal{A}_\alpha]\!]^{\mathbf{b}\otimes\mathbf{d}|_x}$ and $d' \in [\![\mathcal{A}_\beta]\!]^{\mathbf{b}\otimes\mathbf{d}|_x}$. This means that there is a path that starts at x and ends at some position $x{\cdot}y$ that satisfies α, in the sense that whenever a node expression ξ has to hold in a position $x{\cdot}x'$ (where $x' \preceq y$), $\mathcal{A}_\alpha$ verifies that $\xi \in \mathbf{b}(x{\cdot}x')$. By inductive hypothesis, we obtain that $x' \in [\![\xi]\!]^{\mathbf{a}\otimes\mathbf{d}}$. Hence, $(x, x{\cdot}y) \in [\![\alpha]\!]^{\mathbf{a}\otimes\mathbf{d}}$, where $d = \mathbf{d}(x{\cdot}y)$. Applying the same reasoning for β and d', we obtain that $x \in [\![\langle\alpha\odot\beta\rangle]\!]^{\mathbf{a}\otimes\mathbf{d}}$. The case where $\psi = \langle\alpha\rangle$ is only easier. Finally, if ψ is a boolean combination, we simply need to apply the inductive hypothesis and the rules of locally consistent sets.

Suppose now that $\mathbf{t} \models \eta$ for some data tree $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$. We show that $\mathbf{a} \otimes \mathbf{d}$ is accepted by the automaton that results from the translation above. Let $\mathbf{b}(x) = \{\psi \in \{\eta\}^\neg \mid x \in [\![\psi]\!]^{\mathbf{t}}\}$ for every $x \in \mathsf{pos}(\mathbf{t})$. It is easy to check that $\mathbf{b}(x)$ is a locally consistent set and hence that $\mathbf{b}(x) \in \mathbb{B}$. Note that $\mathbf{a} \otimes \mathbf{b}$ is accepted by the transducer $\mathcal{R}$ since $\neg\mathbf{a}(x) \notin \mathbf{b}(x)$. The verifier accepts $\mathbf{b} \otimes \mathbf{d}$ since every test performed at a position x corresponds basically to testing $\mathbf{b}(x) = \{\psi \in \{\eta\}^\neg \mid x \in [\![\psi]\!]^{\mathbf{t}}\}$ by using the labels of the descendant positions. For every $\psi \in \{\eta\}^\neg$ the following can be shown by induction on the size of ψ. If $\psi \in \mathbf{b}(x)$ (in other words, if $x \in [\![\psi]\!]^{\mathbf{t}}$), and ψ is of the form $(\neg)\langle\alpha \odot \beta\rangle$ or $(\neg)\langle\alpha\rangle$, then the formula of the verifier ψ' is true in $\mathbf{t}|_x$, where ψ' is the translation of ψ according to $\mathsf{v}(b)$. This proves that $\mathbf{t}$ is accepted by $(\mathcal{R}, \mathcal{V})$. □

The result above already gives us a decidability procedure for the satisfiability problem. Let us analyse its upper bound.

COROLLARY 6.3. *The translation of Theorem 6.2 yields an automaton that uses at most* $\mathsf{R} = 2$ *relations per existential clause and at most* $\mathsf{V} = 2$ *variables. It can be built in exponential time in* η *such that* $\mathsf{K} + \mathsf{N} \leq p(|\eta|)$ *for some polynomial* p. *The sets of states* $\dot{Q}$ *and* $\tilde{Q}$ *have only one state.*

The proof for the Corollary can be found in the online appendix.

From Theorem 6.2 and Corollary 6.3 we conclude that the satisfiability problem for the full fragment is in ExpTime.

Theorem 6.4 (main result). *SAT-regXPath*$(\downarrow, =)$ *is in* ExpTime.

Proof. Given a formula $\eta \in \mathsf{regXPath}(\downarrow, =)$ we build, in exponential time, a $\mathscr{E}$-DD automaton $(\mathcal{R}, \mathcal{V})$ as in Theorem 6.2. As remarked in Corollary 6.3, $\mathcal{R}$ is such that the set of states $\dot{Q}$ and $\tilde{Q}$ are fixed. $\mathcal{V}$ is such that K and N are polynomial in $|\eta|$, $\mathsf{R} = 2$, and $\mathsf{V} = 2$.

We run the emptiness algorithm of Theorem 4.30 on $(\mathcal{R}, \mathcal{V})$. Since R is fixed, by Corollary 4.31, the time consumed by this algorithm is bounded by

$$(\mathsf{Aut} \cdot |\mathcal{R}| \cdot \mathsf{V} \cdot \mathsf{K} \cdot \mathsf{N})^{p(|\tilde{Q}|, \mathsf{V}, \mathsf{K}, \mathsf{N})}$$

for some polynomial p. Notice that all the variables in the exponent, namely $|\tilde{Q}|, \mathsf{V}, \mathsf{K}, \mathsf{N}$, are bounded by a polynomial in $|\eta|$. The remaining variables can be at most exponential in $|\eta|$. Hence, we have an exponential time algorithm for testing the satisfiability of a formula $\eta \in \mathsf{regXPath}(\downarrow, =)$. □

Lower bound

We next prove the ExpTime-hardness of satisfiability for $\mathsf{XPath}(\downarrow_*, =)$. Remarkably, this logic cannot express a *one step* down in the tree as it does not possess the $\downarrow$ axis, and this will be the major obstacle in the coding.

Theorem 6.5. *SAT*-$\mathsf{XPath}(\downarrow_*, =)$ *is* ExpTime-*hard.*

Proof. The proof is by reduction from the *two-player corridor tiling game.* An instance of this game consists of a set T of tiles $T = \{T_1, \dots, T_s\}$, a special winning tile T_s, the sequence of initial tiles $\{T_1^0, \dots, T_n^0\} \subseteq T$, and the horizontal and vertical tiling relations $H, V \subseteq T \times T$. The game is played on an $n \times \mathbb{N}$ board where the initial configuration of the first row is given by $T_1^0 \cdots T_n^0$. At any moment during the game any pair of horizontally consecutive tiles must be in the relation H and every pair of vertically consecutive tiles in the relation V. The game is played by two players: *Abelard* and *Eloise*. Each player takes turn in placing a tile of his or her choice, filling the board from left to right, from bottom to top, always respecting the horizontal and vertical constraints H and V. *Eloise* is the first to play, and she wins if during the game the winning tile T_s is placed on the board, or if *Abelard* cannot place any tile. Otherwise, if the game continues indefinitely or if *Eloise* cannot place any tile, the game is won by *Abelard*. A *partial* game is a game that may not have finished. We assume that the tile T_s can only appear as the last placed element in the board (and in this case the game is finished). It is known that deciding whether *Eloise* has a winning strategy is ExpTime-complete.

Abstract representation of a winning strategy. It is easy to see that in this game *Eloise* has a winning strategy if, and only if, she has a strategy to win before the row s^n of the board is reached (s is the number of tiles). (Otherwise, there would be a repeated configuration in two different rows, and the game could be shrunk to one with less than s^n rows.) Hence, we represent a partial game as a string of length at most $s^n \cdot n$, containing the plays on the board from left to right, bottom to top, respecting the constraints H and V. We represent a winning strategy for *Eloise* as a tree of nodes labeled with tiles such that

—the root contains T_1^0,

—every node at even depth (*e.g.*, the root) contains as children every tile T such that the path of tiles (from the root to the current node) appended with T is a partial game,

—every node at odd depth with a tile $T \neq T_s$ contains at least one children,

—all the maximal paths of the tree represent winning games for *Eloise*.

Concrete representation of a winning strategy. We must now produce a property of $\mathsf{XPath}(\downarrow_*, =)$ such that any tree that satisfies it represents a winning strategy for *Eloise*. We use the coding of a winning strategy as presented before, extended with some extra nodes and labels, which are necessary to make sure that every path contains at most $n \cdot s^n$ tiles, and that the nodes verify the H and V restrictions. For simplicity, we assume that n is an even number, and hence that all positions in odd columns are played by *Eloise* and the others by *Abelard*. (A similar coding strategy can be used when n is odd.)

Our alphabet consists of

—the symbols $I_1 \dots I_n$ that indicate the current column of the corridor,

—the symbols $b_0 \dots b_m$ where $m = \lceil (n+1) \cdot \log(s) \rceil$ that act as *bits* to count from 0 to s^n (it is enough that they count *at least* up to s^n),

—the symbols $T_1 \dots T_s$ to code the tile placed at each play,

—a symbol # to separate rows, and an extra symbol \$ whose role will be explained later.

Inside a path, each block of nodes between two consecutive occurrences of # codes the evolution of the game for a particular row. Each node labeled I_i has a tile associated, coded as a descendant node with the same data value containing some label T_j as label. For example, in Figure 5, the first column I_1 of the current row is associated to the tile T_3, because $\langle T_3, 1\rangle$ is a descendant of $\langle I_1, 1\rangle$ with the same data value. Similarly, each occurrence of # is associated to a number, which is the number of the current row. This number is coded by the b_i elements with the same data value. In the example, $\langle \#, 0\rangle$ is associated to the bits b_0 and b_2 that give the binary number 101.

Finally, the symbol \$ is used to delimit the region where the next element of the coding must appear, this will be our way of thinking the *next step* of the coding. This is to move from one position of the board to the next one, from the last position to the # delimiter, and from the # to the first position of the next row. Intuitively, between an element I_i, $i < n$, and the element \$ with the same data value, only I_{i+1} may appear. (There may be, however, more than one node with label I_i before the appearance of I_{i+1}, or more than one node I_{i+1}. This corresponds o the fact that we have a reflexive-transitive relation $\downarrow_*$.) This mechanism of coding a very relaxed 'one step' is the building block of our coding. The idea is that since the logic lacks the $\downarrow$ axis, we need to restrict the appearance of the next move of the game to a limited fragment of the tree. By means of this element \$, we can state, for example, that whenever we are in a I_2 element, then in this restricted portion I_3

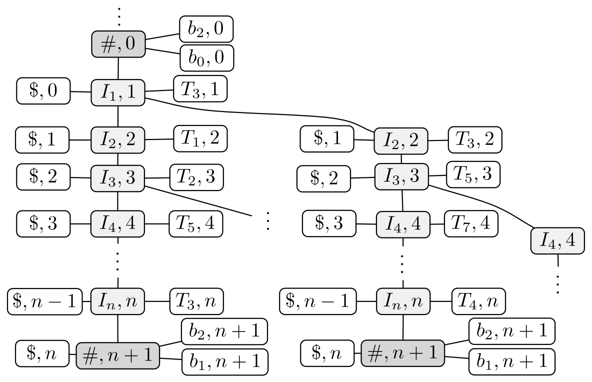

Fig. 5. Part of the model coding all the plays of row 5, which is between the #-element associated to 5 (101 in binary), and the element # with number 6 (110).

must appear with $\langle \varepsilon = \downarrow_*[I_3]\downarrow_*[\$]\rangle$. In a similar way we can demand that there is a prefix of I_2 elements, after which *all* elements have label I_3, until the occurrence of the label \$, as we will see later.

In Figure 5 we show an example of a possible extract of the tree between the # associated to the number 5 until the next # associated to the number 6. The coding forces the tree to have branching as it contains all possible answers of *Abelard* at even positions.

Properties of the tree coding a winning strategy. Since the properties expressed by our logic cannot avoid having repeated consecutives labels along a path, the coding will handle *groups* of nodes with the same label. Let us call an *a-group* to a maximal connected segment of a path that has the label a. The fact that there could be a sequence of elements with equal label does not cause any problem to the coding. In some sense it is redundant information in the coding.

Following the intuition given before, we spell out the concrete properties of the data tree that encodes a winning strategy for *Eloise*.

(1) For every i, we demand that there are no data values shared by different I_i-groups along a path. Likewise for all T_i and #-groups.

(2) The nodes labeled by \$ are leaves, in the sense that no other symbol may appear as descendant.

(3) Every I_i has its corresponding \$, *i.e.*, it has a descendant labeled \$ with the same data value.

(4) Every I_i has a next element, unless it contains the winning tile. That is, there is a I_{i+1} between I_i and its corresponding \$ if $i < n$ (and similarly, a # after I_1, and a I_1 after #).

(5) Each I_i has a unique tile: there is a descendant with equal data value and a tile T as label, and all descendants with equal data value carrying a tile, have the tile T.

(6) Every I_i inside a step along a branch must have the same tile.

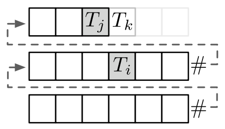

Fig. 6. Every legal move T_k of *Abelard* is played.

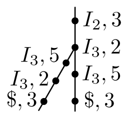

Fig. 7. Repeated elements in the coding.

(7) Between I_i ($i < n$) and its corresponding \$ there can only appear an I_{i+1}-group. (The same applies for the labels I_n and #, and for # and I_1.)

(8) The tiles match horizontally: the tiles corresponding to any two consecutive nodes labeled I_i and I_{i+1} are in the H relation. Likewise, the tiles match vertically: the tiles corresponding to any two nodes labeled I_i separated by exactly one #-group are in the V relation.

(9) All the elements corresponding to the *first* row have tiles $T^0_1 \cdots T^0_n$.

(10) Every legal move of *Abelard* is taken into account. For every node I_i with i being odd such that there is a tile T_k that can be played in the next position (according to H, V and the tiles already placed, as in Figure 6), then it must appear in the next position. Note that this forces a branching in the tree.

(11) The data value of a # element is associated to a counter. The least significant bit corresponds to b_0. A bit 1 at a bit position i is coded as the presence of a descendant node with the same data value labeled b_i. The counter starts in 0, and along a path, each time a #-group appears, the counter increments by one.

(12) There is no # element that has all the b_i bits in 1. Because that would mean that *Eloise* was not able to put a T_s tile in less than s^n rounds.

As already mentioned, we do not avoid having more than one element before the \$. As shown in Figure 7, there may be consecutive repetitions of the same label along a path, or subtrees that are duplicated, but this does not spoil the coding. We are actually forcing properties for *all* branches and all possible extra elements that the tree may contain. Any extra element or branching induces more copies of winning strategies for *Eloise*.

For every data tree with these properties, one can replace all consecutive appearances of the same label along a path by only one appearance, and strip off all the nodes containing labels which are not tiles. This gives us a winning strategy. Conversely, given a winning strategy for *Eloise*, we can add data values to the tree and the necessary nodes to transform it into a data tree that satisfies all the aforementioned properties.

Enforcing the properties in $\mathsf{XPath}(\downarrow_*, =)$. We first define some predicates that we will use throughout the coding. $\mathsf{s}^k_\sigma(\varphi)$ evaluates φ at a node at k-steps (with our way

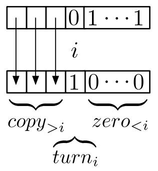

Fig. 8. A bitwise increment of the counter.

of coding a step as we have seen before) from the current point of evaluation, given that the current symbol is σ. For this purpose we first define $next(I_i) := I_{i+1}$ (if $i < n$), $next(I_n) := \#$ and $next(\#) := I_1$. Hence, we define for $a \in \{\#, I_1, \ldots, I_n\}$,

$$\mathsf{s}^0_a(\varphi) := a \wedge \varphi \qquad \mathsf{s}^{k+1}_a(\varphi) := a \wedge \langle \varepsilon = {\downarrow_*}[\mathsf{s}^k_{next(a)}(\varphi)]{\downarrow_*}[\$]\rangle$$

Similarly, t_i checks that the tile of the current node I corresponds to T_i,

$$\mathsf{t}_i := \langle \varepsilon = {\downarrow_*}[T_i]\rangle$$

bit_i checks that the i-bit of the counter's binary encoding of a #-node is one (1),

$$bit_i := \langle \varepsilon = {\downarrow_*}[b_i]\rangle$$

and G forces a property to hold at all nodes of the tree.

$$\mathsf{G}(\varphi) := \neg\langle {\downarrow_*}[\neg\varphi]\rangle$$

We now exhibit the XPath formulæ for each of the properties just seen.

(1) For every I_i: $\neg{\downarrow_*}[I_i \wedge \langle \varepsilon = {\downarrow_*}[\neg I_i]{\downarrow_*}[I_i]\rangle]$. Likewise for T_i and #.

(2) $\mathsf{G}(\$ \to \neg\langle{\downarrow_*}[\neg\$]\rangle)$.

(3) For every I_i: $\mathsf{G}(I_i \to \langle \varepsilon = {\downarrow_*}[\$]\rangle)$.

(4) $\mathsf{G}(I_i \wedge \neg\mathsf{t}_s \to \mathsf{s}^1_{I_i}(\top)) \wedge \mathsf{G}(\# \to \mathsf{s}^1_{\#}(\top))$.

(5) For every I_i and $\ell \neq j$: $\mathsf{G}(\neg(\mathsf{t}_\ell \wedge \mathsf{t}_j))$. For every i: $\mathsf{G}(I_i \to \bigvee_j \mathsf{t}_j)$.

(6) For every $i < n$ and $j \neq k$, $\mathsf{G}(I_i \to \neg\langle \varepsilon = {\downarrow_*}[I_{i+1} \wedge \mathsf{t}_j]{\downarrow_*}[I_{i+1} \wedge \mathsf{t}_k]{\downarrow_*}[\$]\rangle)$. (And a similar condition for # and I_1.)

(7) For every $i < n$ and $a \notin \{I_i, I_{i+1}, \$\}$, $\mathsf{G}(I_i \to \neg\langle \varepsilon = {\downarrow_*}[a]{\downarrow_*}[\$]\rangle)$, $\mathsf{G}(I_i \to \neg\langle \varepsilon = {\downarrow_*}[I_{i+1}]{\downarrow_*}[I_i]{\downarrow_*}[\$]\rangle)$, and $\mathsf{G}(I_i \to \neg\langle \varepsilon = {\downarrow_*}[\$]{\downarrow_*}[I_i \vee I_{i+1}]{\downarrow_*}[\$]\rangle)$.

(8) The tiles match horizontally: for every k and T_i, T_j such that $(T_i, T_j) \notin H$, $\neg\langle{\downarrow_*}[I_k \wedge \mathsf{t}_i \wedge \mathsf{s}^1_{I_k}(\mathsf{t}_j)]\rangle$. The tiles match vertically: for every k and T_i, T_j such that $(T_i, T_j) \notin V$, $\neg\langle{\downarrow_*}[I_k \wedge \mathsf{t}_i \wedge \mathsf{s}^{n+1}_{I_k}(\mathsf{t}_j)]\rangle$.

(9) For all $i \in [1..n]$ and tile $T_j = T^0_i$, $\mathsf{s}^i_{\#}(\mathsf{t}_j)$ must hold at the root.

(10) For every T_i, T_j, T_k such that $(T_i, T_k) \in V$ and $(T_j, T_k) \in H$:

$$\neg\big\langle{\downarrow_*}\big[I_{2\ell} \wedge \mathsf{t}_i \wedge \mathsf{s}^n_{I_{2\ell}}\big(I_{2\ell-1} \wedge \mathsf{t}_j \wedge \neg\mathsf{s}^1_{I_{2\ell-1}}(\mathsf{t}_k)\big)\big\rangle$$

(11) It is easy to code that the first # is all-zero. The increment of the counter between two # is coded as in Figure 8, by $\mathsf{G}(\# \wedge \mathit{flip}(i) \to zero_{<i} \wedge turn_i \wedge$

$copy_{>i}$), where

$$flip(i) := \neg bit_i \wedge \bigwedge_{j<i} bit_j \quad zero_{<i} := \bigwedge_{j<i} \neg \mathsf{s}_{\#}^{n+1}(bit_j) \quad turn_i := \neg \mathsf{s}_{\#}^{n+1}(\neg bit_i)$$

$$copy_{>i} := \bigwedge_{j>i} (bit_j \wedge \neg \mathsf{s}_{\#}^{n+1}(\neg bit_j)) \vee (\neg bit_j \wedge \neg \mathsf{s}_{\#}^{n+1}(bit_j))$$

(12) $\mathsf{G}(\# \to \neg \bigwedge_i bit_i)$

This completes the coding. It is easy to see that all the formulæ have polynomial size on s and n, and that they express the previous properties. Hence, *Eloise* has a winning strategy in the two-player corridor tiling game iff the conjunction of the formulæ just described is satisfiable. Notice that the above reduction does not use path unions, and this means that even XPath($\downarrow_*, =$) stripped of path unions is ExpTime-hard. □

6.2 PSpace fragments

We now turn to some other downward fragments of XPath. We complete the picture by analysing the complexity of all the possible combinations of downward axes in the presence or absence of data values tests. We first introduce a basic definition that we use throughout the section.

Definition 6.6. We say that the logic $\mathscr{P}$ has the **poly-depth model property** if there exists a polynomial p such that for every formula $\varphi \in \mathscr{P}$, φ is satisfiable if and only if φ is satisfied by a data tree of height at most $p(|\varphi|)$.

We can now prove the following statement that we will later use to show PSpace-completeness for XPath($\downarrow, =$).

Proposition 6.7. *Every fragment $\mathscr{P}$ of regXPath($\downarrow, =$) that has the poly-depth model property is in* PSpace.

The proof of this Proposition can be found in the online appendix.

We use the result above to prove the following proposition, whose proof can be found in the online appendix.

Proposition 6.8. *SAT-XPath($\downarrow, =$) is* PSpace*-complete.*

This concludes our analysis of downward fragments of XPath with data tests. Summing up, we showed that the satisfiability problem for regXPath($\downarrow, =$), XPath($\downarrow, \downarrow_*, =$) and XPath($\downarrow_*, =$) is ExpTime-complete, while it is PSpace-complete for XPath($\downarrow, =$). For the sake of completeness, we now turn to downward fragments where no data tests are available.

Corollary 6.9. *SAT-XPath($\downarrow$) is* PSpace*-complete.*

Proof. The lower bound by [Benedikt et al. 2008, Theorem 5.1] and the upper bound by Proposition 6.8. □

Proposition 6.10. XPath($\downarrow_*$) *is* PSpace*-hard.*

The proof goes by reduction from an instance of the QBF (Quantified Boolean Formula [Garey and Johnson 1979]) validity problem to SAT-XPath($\downarrow_*$) and can be found in the online appendix.

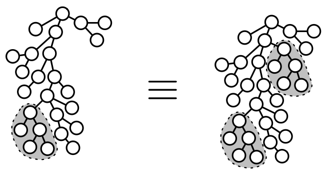
Fig. 9. The subtree copy property.

Now we focus in finding an upper bound for SAT-$\mathsf{XPath}(\downarrow_*)$. We prove that it is in PSpace by the poly-depth model property. But before doing that, we need to introduce an important property of this logic. The **subtree copy** property states that if a tree satisfies some $\mathsf{XPath}(\downarrow_*)$ property at the root, then the tree where some subtree was copied at a higher position of the tree (as depicted in Figure 9) also satisfies the property.

Lemma 6.11 (subtree copy). *Given a tree* $\mathbf{t}$*, and given two positions* $x, y \in \mathsf{pos}(\mathbf{t})$ *with* $x \prec y$*, consider the last index* l *such that* $x{\cdot}l \in \mathsf{pos}(\mathbf{t})$*. Let* $\mathbf{t}'$ *be defined as follows.*

$$\mathsf{pos}(\mathbf{t}') = \mathsf{pos}(\mathbf{t}) \cup \{x{\cdot}(l+1){\cdot}z \mid y{\cdot}z \in \mathsf{pos}(\mathbf{t})\}$$

$$\mathbf{t}'(z) = \begin{cases} \mathbf{t}(z) & \textit{if } z \in \mathsf{pos}(\mathbf{t}) \\ \mathbf{t}(y{\cdot}w) & \textit{if } z = x{\cdot}(l+1){\cdot}w \end{cases}$$

Then, for every $w \in \mathsf{pos}(\mathbf{t})$ *and* $\varphi \in \mathsf{XPath}(\downarrow_*)$*,* $\mathbf{t}|_w \models \varphi$ *iff* $\mathbf{t}'|_w \models \varphi$*. The shapes of* $\mathbf{t}$ *and* $\mathbf{t}'$ *are illustrated in Figure 9.*

The proof of this Lemma can be found in the online appendix.

Note that the preceding Lemma 6.11 is a stronger property than that of Proposition 4.9, but this one holds only for $\mathsf{XPath}(\downarrow_*)$, a logic with no data tests or $\downarrow$ axis. Having stated the subtree copy property, we can now show the following proposition.

Proposition 6.12. *SAT-*$\mathsf{XPath}(\downarrow_*)$ *is in* PSpace.

Indeed it can be shown that $\varphi \in \mathsf{XPath}(\downarrow_*)$ is satisfiable iff it is satisfied by a tree of height bounded by $|\varphi|^2$. The full proof can be found in the online appendix.

We then have as a corollary from Proposition 6.12 and Proposition 6.10 that $\mathsf{XPath}(\downarrow_*)$ is complete for PSpace.

Theorem 6.13. *SAT-*$\mathsf{XPath}(\downarrow_*)$ *is* PSpace*-complete.*

So far we have that, in the presence of data values, the presence of the descendant axis $\downarrow_*$ produces an increase (in the case PSpace $\neq$ ExpTime) in the complexity from PSpace to ExpTime. However, we argue that it is not the ability to test for data equality of distant elements what produces this increase in complexity. It is, as a matter of fact, in the ability to test data values against that of the root in formulæ like $\langle \varepsilon = \downarrow_*[a] \rangle$. We show that if we remove this kind of data tests, the resulting logic is in PSpace, even though the fragment contains the $\downarrow_*$ axis.

Definition 6.14. We denote by $\mathsf{XPath}^{\not\varepsilon}(\downarrow_*, =)$ the fragment of $\mathsf{XPath}(\downarrow_*, =)$ where ε path formulæ are forbidden, and in general where there is no ε-testing on a path

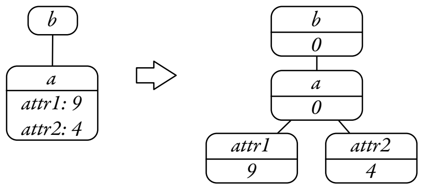

Fig. 10. Transformation from an XML to a data tree.

(like in $[\varphi]\downarrow_*$), that is, such that path formulæ are defined

$$\alpha \ ::= \ \downarrow_* \ \mid \alpha[\varphi] \mid \alpha\beta \mid \alpha \cup \beta \ .$$

PROPOSITION 6.15. *SAT*-$\mathsf{XPath}^{\not\epsilon}(\downarrow_*, =)$ *is* PSPACE-*complete.*

The proof of the upper-bound goes by showing the poly-depth model property, and can be found in the online appendix.

6.3 XML documents vs. data trees

As outlined before, XML documents may have multiple attributes with data values on each element, while data trees can only have one. Here we will show that every result we have stated in terms of data trees also holds on the class of XML documents. Let us consider that the finite set of symbols is partitioned between the names for attributes and the symbols of the XML elements, $\mathbb{A}_{attr} \cup \mathbb{A}_{elem}$.

We define the class of **attributes data trees** as the trees $\mathbf{a} \otimes \mathbf{d}$ where every position carries one label from $\mathbb{A}_{elem}$ and many data values indexed by $\mathbb{A}_{attr}$ that we call 'attributes', $\mathbf{a} : P \to \mathbb{A}_{elem}$, $\mathbf{d} : P \to \wp_{<\infty}(\mathbb{A}_{attr} \times \mathbb{D})$[8] for some $P \in \mathsf{TreesPos}$. It follows that any XML document can be seen as an attributes data tree.

Let us consider then the extension of the languages where different attributes may be compared, where node expressions are defined

$$\varphi \ ::= \ a \mid \neg\varphi \mid \varphi \wedge \psi \mid \langle\alpha\rangle \mid \langle \alpha@\mathit{attr1} \odot \beta@\mathit{attr2}\rangle$$

where $\odot \in \{=, \neq\}, a \in \mathbb{A}_{elem}$ and $\mathit{attr1}, \mathit{attr2} \in \mathbb{A}_{attr}$. Let us call this logic attrXPath. This language with the expected semantics over attributes data trees can encode any attrXPath request on an XML document.

SAT-attrXPath($\downarrow, \downarrow_*, =$) is EXPTIME-easy

As already mentioned, each XML document can be coded in a data tree by adding one child for each attribute with its corresponding value as in Figure 10. We can enforce this property with XPath($\downarrow_*, \downarrow, =$), by stating that all the nodes with a symbol from $\mathbb{A}_{attr}$ are leaves,

$$\varphi_{struct} = \neg\langle \downarrow_*[\bigvee_{s \in \mathbb{A}_{attr}} s \ \wedge \langle\downarrow\rangle] \rangle \ .$$

We can interpret any attrXPath formula as an XPath formula by considering an extended alphabet $\mathbb{A} = \mathbb{A}_{elem} \cup \mathbb{A}_{attr}$ and replacing every appearance of '$@attr1$' by '$\downarrow [attr1]$'. Let us call tr to this translation.

[8]Note that in this model an attribute may be associated with multiple data values.

We can then decide the satisfiability of a formula ψ of $\mathsf{attrXPath}(\downarrow, \downarrow_*, =)$ on attributes data trees by testing the satisfiability of '$tr(\psi) \wedge \varphi_{struct}$' on data trees. Since we have that $\mathsf{XPath}(\downarrow, \downarrow_*, =)$ is in ExpTime, we also have an ExpTime decidability procedure for the full downward fragment of attrXPath (as the translation tr is clearly performed in PTime) even with the Kleene star operator.

SAT-attrXPath($\downarrow_*, =$) is ExpTime-hard

On the other hand, any XPath formula on data trees can be thought of an attrXPath formula that uses at most one attribute. We can then deduce the ExpTime-hardness result of $\mathsf{attrXPath}(\downarrow_*, =)$ from that of $\mathsf{XPath}(\downarrow_*, =)$.

SAT-attrXPath($\downarrow, =$) is PSpace-complete

For the case of $\mathsf{attrXPath}(\downarrow, =)$ we can do the same translation, the only difference being that for a formula $\psi \in \mathsf{attrXPath}(\downarrow, =)$ the structure can be forced by

$$\varphi_{struct} = \bigwedge_{0 \leq n \leq d+1} \neg\langle\downarrow^n [\bigvee_{s \in \mathbb{A}_{attr}} s \wedge \langle\downarrow\rangle]\rangle$$

where d is the maximum quantity of nested occurrences of $\downarrow$ in ψ. It is easy to see that this forces the requested property for all the portion of the data tree that we are interested in. That is, for the whole region that $tr(\psi)$ can access. This is associated with the poly-depth model property of the logic. We then have that $\mathsf{attrXPath}(\downarrow, =)$ is PSpace-complete.

6.4 In the presence of regular languages

In this section we make some observations on the following problem for $\mathscr{L}$ a class of tree languages and $\mathscr{P}$ a fragment of XPath.

SAT-$\mathscr{P} + \mathscr{L}$	Satisfiability problem for $\mathscr{P}$ under $\mathscr{L}$
Input:	$\varphi \in \mathscr{P}$ and an automaton representing a language $\mathcal{L} \in \mathscr{L}$.
Output:	Is there a tree $\mathbf{t} \in \mathcal{L}$ such that $\mathbf{t} \models \varphi$?

By [Figueira 2010a] we know that satisfiability of downward XPath under a regular language is decidable. However, the problem has a very big complexity. Even for the fragment containing only the descendant axis, it can be shown that there is no algorithm that solves the satisfiability problem under a regular language in primitive recursive time or space [Figueira and Segoufin 2009]. However, if the language to be tested at the siblinghoods is restricted to be extensible, we can translate this problem to the emptiness problem for DD automata obtaining the following result.

Theorem 6.16. *SAT-*$\mathsf{regXPath}(\downarrow, =) + \mathscr{E}_{tree}$*, that is, the satisfiability problem for* $\mathsf{regXPath}(\downarrow, =)$ *under the class* $\mathscr{E}_{tree}$ *of extensible tree regular languages, is in* ExpTime.

Proof. Let $\varphi \in \mathsf{regXPath}(\downarrow, =)$ and $\mathcal{L} \in \mathscr{E}_{tree}$, represented as an $\mathscr{E}$-transducer $\mathcal{R}_{\mathcal{L}}$ such that $\mathcal{R}_{\mathcal{L}}(\mathbf{t} \otimes \mathbf{t})$ iff $\mathbf{t} \in \mathcal{L}$.

We first build the DD automaton $(\mathcal{R}, \mathcal{V})$ resulting from the translation of φ by Theorem 6.2. We compute the composition $\mathcal{R}' = \mathcal{R}_{\mathcal{L}} \circ \mathcal{R}$ such that $\mathcal{R}(\mathbf{t} \otimes \mathbf{t}')$ iff $\mathcal{R}_{\mathcal{L}}(\mathbf{t} \otimes \mathbf{t})$ and $\mathcal{R}(\mathbf{t} \otimes \mathbf{t}')$ in polynomial time in $\mathcal{R}$ and $\mathcal{R}_{\mathcal{L}}$. We then test the emptiness of $(\mathcal{R}', \mathcal{V})$ in exponential time in the number of states $|\tilde{Q}|$ of the

```
<!ELEMENT book_list (book*)>
<!ELEMENT book ((author, birthdate?)+, chapter+)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT birthdate (#PCDATA)>
<!ELEMENT chapter (#PCDATA)>
```

Table III. Example of a DTD under which regXPath($\downarrow, =$) is decidable in ExpTime.

automata corresponding to the (extensible) languages from the transitions of $\mathcal{R}'$. The resulting reduction is in ExpTime because the same arguments used to show 6.4 can be applied. □

The above Theorem 6.16 implies that the satisfiability of regXPath($\downarrow, =$) under certain kind of restrictions is decidable in ExpTime. Note that these restrictions may be specified as DTD, XML Schema, Relax NG, etc. For example, it would mean that this logic is decidable under DTD whose every type is defined under some transitive operator $+$ or $*$. That is, that the definition of every type is in REG_* (as defined in Definition 3.7), like in the example of Table III.

7. CONCLUDING REMARKS

In Section 3 we introduced an automata model over data trees. Here we explore very briefly two natural extensions that could be added to this automaton. Firstly, the possibility of allowing *any* alternation free first order formula in the set Φ of properties that the verifier can test, instead of the restricted kind where no negation of a relation may occur. Secondly, what happens if we allow constants in Φ.

Negation of relations

The reader may have noticed in Definition 3.2 that the DD automata model does not allow to have negated appearances of a relation D_i under a (positive) existential quantification. This is not by chance, and in effect we can see that if we allow to have arbitrary boolean combinations of D_i relations we fall into a much harder emptiness problem. Although the decidability of the resulting model is not clear, it is possible to show that in the case it is decidable, the emptiness problem cannot be solved in primitive recursive complexity.

Consider the simple formula $\neg(\exists x.\neg D_1(x) \wedge D_2(x))$, such that $\mathcal{A}_1$ recognizes $\{b\,b \mid b \in \mathbb{B}\}$ and $\mathcal{A}_2$ recognizes $\{b\,b\,b \mid b \in \mathbb{B}\}$. In other words, $\mathcal{A}_1$ simply goes to any child of the root, and $\mathcal{A}_2$ goes to any grandchild of the root. This kind of property intuitively tests that all the data values appearing at depth $l+1$ also appear at depth l. Although we will not enter into detail, it is possible to code a run of a weak version of a n-counters Minsky machine with increment errors (also called *gainy counter machine*, or *incrementing counter automaton*, *cf.* [Demri and Lazić 2009]) by using this kind of property. It is known that the emptiness problem for these kind of machines is non-primitive recursive [Schnoebelen 2002], and hence the emptiness problem for this extended automaton cannot be primitive recursive.

Constants

Another simple extension that DD automata may allow to have is the fact of having a set of constant data values. This does not change the complexity results. The results and definitions of Sections 4.1, 4.3, 4.2 remain valid. In Section 4.4

we modify the tree configurations by keeping explicit track of these constants, by adding the description of these constants to the data description mapping α at all configurations, and modifying the conditions of $\vdash$ accordingly. This implies that we can verify the satisfiability of $\mathsf{regXPath}(\downarrow, =)$ with constants also in ExpTime.

Discussion

We have shown the complexity of various downward fragments of XPath, as summarized in Table IV. The highest complexity class we obtained is ExpTime. In the presence of data equality tests, is a well-behaved fragment considering that in the presence of all the axes XPath is undecidable. One important reason for this is the absence of any sibling axis. Indeed, as soon as any horizontal navigation is allowed in the logic, the problem becomes non-primitive recursive. However, we have shown that we can evaluate some restricted fragment of XML Schema or DTD that cannot limit the quantity of occurrences of nodes of a certain type, but that can verify that there is a certain structure in the siblings of the tree. For example, we can express that the children of every node with label book form a sequence of labels in the language $(\mathsf{author}\,(\mathsf{chapter})^*)^+$ (since it is an extensible language). Also, by solving the satisfiability problem we are also able to solve the containment and equivalence problems of node expressions for free, since we work with logics closed under boolean operators. We leave open the question of whether the inclusion of path expressions (as binary relations) is also decidable in ExpTime.

We introduced the new class of Downward Data automata that capture all the expressivity of $\mathsf{regXPath}(\downarrow, =)$. This automata model is more expressive than XPath. It can test properties like, for example, that there are exactly 7 data values with label book; or that every node labeled book has between 1 and 4 children author with different data value; or that there is a data value that can be simultaneously accessed by three different branches, satisfying the path expressions "$\downarrow_*[\mathsf{article}]\downarrow[\mathsf{author}]$", "$\downarrow_*[\mathsf{conference}]\downarrow[\mathsf{chair}]$", and "$\downarrow_*[\,\mathsf{scientist}\wedge\langle\,(\downarrow[\mathsf{adviser}])^*\downarrow[\mathsf{sex}]\downarrow[\mathsf{female}]\,\rangle\,]\downarrow[\mathsf{name}]$".

By the proof of decidability of the DD automata, we conclude that there is a normal form of the model for downward XPath. If a formula $\eta \in \mathsf{regXPath}(\downarrow, =)$ is satisfiable, then it is satisfiable in a model of exponential height and polynomial branching width, whose data values are such that only a polynomial number of data values can be shared between any two disjoint subtrees. This property is reflected by the fact that the emptiness of the automaton that results from the translation of a downward XPath formula only depends on a polynomial number of data values at every position. However, there is no syntactic restriction in the automaton, it can retrieve and compare any number of data values between them and the root's data value at each step of its execution.

It would also be interesting to study if the techniques of this work could be used to treat the satisfiability problem of monadic datalog programs [Ceri et al. 1989] extended with equality and inequality of data values.

Finally, a recent result [Figueira 2011] on XPath on data words, suggests that it is plausible that $\mathsf{XPath}(\downarrow, \downarrow_*, \rightarrow^*, {}^*\!\leftarrow, =)$ or even $\mathsf{XPath}(\downarrow, \downarrow_*, \uparrow^*, \rightarrow^*, {}^*\!\leftarrow, =)$ is decidable in elementary time.

Conjecture 7.1 [Figueira 2011, Conjecture 1]. *The satisfiability problem for* $\mathsf{XPath}(\downarrow, \downarrow_*, \rightarrow^*, {}^*\!\leftarrow, =)$ *is decidable in elementary time.*

$\downarrow$	$\downarrow^*$	$=$	**Complexity**	**Details**
●			PSpace-complete	Cor. 6.9
	●		PSpace-complete	Thm. 6.13
●	●		ExpTime-complete	[Marx 2004]
●		●	PSpace-complete	Prop. 6.8
	●	●	ExpTime-complete	Thm. 6.4, Thm. 6.5
●	●	●	ExpTime-complete	Thm. 6.4, Thm. 6.5
regXPath($\downarrow, =$)			ExpTime-complete	Thm. 6.4, Thm. 6.5
XPath$^{\not\epsilon}$($\downarrow_*, =$)			PSpace-complete	Prop. 6.15
regXPath($\downarrow, =$) + $\mathscr{E}_{tree}$			ExpTime-complete	Thm. 6.16, Thm. 6.5

Table IV. Summary of results. All the bounds also hold in the absence of path unions.

Conjecture 7.2 [Figueira 2011, Conjecture 2]. *The satisfiability problem for* XPath($\downarrow, \downarrow_*, \uparrow^*, \rightarrow^*, {}^*\!\leftarrow, =$) *is decidable in elementary time.*

This would be an extension of the result of the present work, and in contrast with the fact that XPath($\downarrow, \downarrow_*, \rightarrow^*, {}^+\!\leftarrow, =$) is undecidable and XPath($\downarrow, \downarrow_*, \rightarrow^+, =$) has non-primitive recursive complexity [Figueira and Segoufin 2009].

REFERENCES

Benedikt, M., Fan, W., and Geerts, F. 2008. XPath satisfiability in the presence of DTDs. *Journal of the ACM 55,* 2, 1–79.

Blackburn, P., de Rijke, M., and Venema, Y. 2001. *Modal logic.* Cambridge University Press.

Bojańczyk, M., Muscholl, A., Schwentick, T., and Segoufin, L. 2009. Two-variable logic on data trees and XML reasoning. *Journal of the ACM 56,* 3, 1–48.

ten Cate, B. 2006. The expressivity of XPath with transitive closure. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'06).* ACM Press, 328–337.

ten Cate, B. and Segoufin, L. 2008. XPath, transitive closure logic, and nested tree walking automata. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'08).* ACM Press, 251–260.

Ceri, S., Gottlob, G., and Tanca, L. 1989. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng. 1,* 1, 146–166.

Clark, J. and DeRose, S. 1999. XML path language (XPath). Website. W3C Recommendation. `http://www.w3.org/TR/xpath`.

Demri, S. and Lazić, R. 2009. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic 10,* 3.

Fan, W., Chan, C. Y., and Garofalakis, M. N. 2004. Secure XML querying with security views. In *ACM SIGACT-SIGMOD-SIGART International Conference on Management of Data (SIGMOD'04).* ACM Press, 587–598.

Figueira, D. 2009. Satisfiability of downward XPath with data equality tests. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'09).* ACM Press, 197–206.

Figueira, D. 2010a. Forward-XPath and extended register automata on data-trees. In *International Conference on Database Theory (ICDT'10).* ACM Press, 230–240.

Figueira, D. 2010b. Reasoning on words and trees with data. Ph.D. thesis, Laboratoire Spécification et Vérification, ENS Cachan, France.

Figueira, D. 2011. A decidable two-way logic on data words. In *Annual IEEE Symposium on Logic in Computer Science (LICS'11)*. IEEE Computer Society Press.

Figueira, D. and Segoufin, L. 2009. Future-looking logics on data words and trees. In *International Symposium on Mathematical Foundations of Computer Science (MFCS'09)*. LNCS, vol. 5734. Springer, 331–343.

Figueira, D. and Segoufin, L. 2011. Bottom-up automata on data trees and vertical XPath. In *International Symposium on Theoretical Aspects of Computer Science (STACS'11)*. Springer, 93–104.

Garey, M. and Johnson, D. 1979. *Computers and intractability: a guide to the theory of NP-completeness*. A series of books in mathematical sciences. W. H. Freeman.

Geerts, F. and Fan, W. 2005. Satisfiability of XPath queries with sibling axes. In *International Symposium on Database Programming Languages (DBPL'05)*. Lecture Notes in Computer Science, vol. 3774. Springer, 122–137.

Gottlob, G., Koch, C., and Pichler, R. 2002. Efficient algorithms for processing xpath queries. In *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB '02. VLDB Endowment, 95–106.

Gottlob, G., Koch, C., and Pichler, R. 2005. Efficient algorithms for processing XPath queries. *ACM Transactions on Database Systems 30,* 2, 444–491.

Jurdziński, M. and Lazić, R. 2008. Alternating automata on data trees and XPath satisfiability. *Computing Research Repository (CoRR)*.

Ladner, R. E. 1977. The computational complexity of provability in systems of modal propositional logic. *SIAM Journal on computing 6,* 3, 467–480.

Martens, W. and Neven, F. 2007. Frontiers of tractability for typechecking simple xml transformations. *J. Comput. Syst. Sci. 73,* 3, 362–390.

Marx, M. 2004. XPath with conditional axis relations. In *International Conference on Extending Database Technology (EDBT'04)*. Lecture Notes in Computer Science, vol. 2992. Springer, 477–494.

Marx, M. 2005. First order paths in ordered trees. In *International Conference on Database Theory (ICDT'05)*. Springer, 114–128.

Schnoebelen, P. 2002. Verifying lossy channel systems has nonprimitive recursive complexity. *Information Processing Letters 83,* 5, 251–261.

Received May 2010; revised May 2011; accepted November 2011

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library by visiting the following URL: `http://www.acm.org/pubs/citations/journals/tocl/2000-0-0/p1-XXXXX`.

THIS DOCUMENT IS THE ONLINE-ONLY APPENDIX TO:

Decidability of downward XPath

DIEGO FIGUEIRA
INRIA, ENS Cachan, LSV & University of Edinburgh

ACM Transactions on Computational Logic, Vol. 0, No. 0, 00 2000, Pages 1–45.

A. MISSING PROOFS

PROOF OF PROPOSITION 3.5. First observe that the set of properties Φ of any verifier is closed under conjunction and disjunction, since it is closed under the operators $\wedge$, $\vee$.

Suppose we have two DD automata $(\mathcal{R}^1, \mathcal{V}^1)$ and $(\mathcal{R}^2, \mathcal{V}^2)$, $\mathcal{R}^i \subseteq \mathit{Trees}(\mathbb{A}\times\mathbb{B}_i)$, $\mathcal{V}^i \subseteq \mathit{Trees}(\mathbb{B}_i\times\mathbb{D})$. We build the intersection automaton $(\mathcal{R}, \mathcal{V})$ where $\mathcal{R} \subseteq \mathit{Trees}(\mathbb{A}\times(\mathbb{B}_1\times\mathbb{B}_2))$ is a transducer that tags each position with a pair of letters, such that $\mathbf{a}\otimes\mathbf{b}_1\otimes\mathbf{b}_2 \in \mathcal{R}$ if, and only if, $\mathbf{a}\otimes\mathbf{b}_1 \in \mathcal{R}^1$ and $\mathbf{a}\otimes\mathbf{b}_2 \in \mathcal{R}^2$. This can be done since $\mathscr{C}$ is closed under componentwise product, as follows. We define the state space as $\dot{Q}^1\times\dot{Q}^2$ and the final states as $\dot{Q}^1_F\times\dot{Q}^2_F$. For every transition $(\dot{q}_1, a, b_1, \mathcal{L}_1)$ of $\mathcal{R}^1$ and $(\dot{q}_2, a, b_2, \mathcal{L}_2)$ of $\mathcal{R}^2$ we have a transition $((\dot{q}_1, \dot{q}_2), a, (b_1, b_2), \mathcal{L}_1 \times_c \mathcal{L}_2)$ in $\mathcal{R}$.

On the other hand we build $\mathcal{V} \subseteq \mathit{Trees}((\mathbb{B}_1\times\mathbb{B}_2)\times\mathbb{D})$ such that $\mathbf{b}_1\otimes\mathbf{b}_2\otimes\mathbf{d} \in \mathcal{V}$ iff $\mathbf{b}_1\otimes\mathbf{d} \in \mathcal{V}^1$ and $\mathbf{b}_2\otimes\mathbf{d} \in \mathcal{V}^2$. This can be done since Φ is closed under conjunction. $\square$

PROOF OF PROPOSITION 3.6. Let $(\mathcal{R}, \mathcal{V})$ with $\mathcal{R} \subseteq \mathit{Trees}(\mathbb{A}\times\mathbb{B})$, $\mathcal{V} \subseteq \mathit{Trees}(\mathbb{B}\times\mathbb{D})$, where $\dot{Q}$ is the set of states of $\mathcal{R}$. We build $(\mathcal{R}^c, \mathcal{V}^c)$ the complement of $(\mathcal{R}, \mathcal{V})$. We define $\mathcal{R}^c \subseteq \mathit{Trees}(\mathbb{A}\times\mathbb{B}')$ with $\mathbb{B}' = \wp(Z)$ where $Z = \dot{Q}\times\mathbb{B}\times\{\mathit{below}, \mathit{notyet}, \mathit{here}\}$. Every node x of the tree $\mathbf{t}$ is labeled by a set of tuples $(\dot{q}, b, i) \in Z$ describing, for each of the possible partial runs of $\mathcal{R}$ on $\mathbf{t}|_x$, whether there is a node that does *not* verify a property demanded by $\mathcal{V}$. More precisely, $\dot{q}$ is the state of the run at the root, b refers to the output letter of the root, and $i \in \{\mathit{below}, \mathit{notyet}, \mathit{here}\}$ is to keep track of whether in the currently described run there is a node falsifying a property demanded by $\mathcal{V}$. The value *notyet* denotes that all the nodes in the subtree $\mathbf{t}|_x$ for this run verify the properties of $\mathcal{V}$; *below* that there is a node different from x that does not verify the property of $\mathcal{V}$, and *here* that in the current run x does not verify the property of $\mathcal{V}$. Formally, $\mathcal{R}^c$ tags every node x of the tree $\mathbf{t}$ with the set of tuples $(\dot{q}, b, i) \in Z$ such that there is a partial run of $\mathcal{R}$ on $\mathbf{t}|_x$ where

—the state of the root is $\dot{q}$,

—the output label of the root is b, and

—$i = \mathit{below}$ iff there is a node of $\mathbf{t}|_x$ different from the root with output label $(\dot{q}', b', i')$ such that $i' = \mathit{here}$.

Further, $\mathcal{R}^c$ checks that the output label of the root contains only tuples of the form $(\dot{q}, b, i)$ where either $i = \mathit{below}$, $i = \mathit{here}$ or $\dot{q}$ is not final in $\mathcal{R}$. This ensures that all possible accepting runs of $\mathcal{R}$ lead to trees that falsify some demand of $\mathcal{V}$ at some node. Notice that if there is a node different from the root with $i' = \mathit{here}$ then $i = \mathit{below}$, and if there is no node different to the root with $i' = \mathit{here}$, then i may be *here* or *notyet*. Also, notice that the state at the leaves has $i = \mathit{here}$ or $i = \mathit{notyet}$ (*i.e.*, no *below*).

Assuming we can build $\mathcal{R}^c$ with the described behavior, we define $\mathcal{V}^c \subseteq \mathit{Trees}(\mathbb{B}' \times \mathbb{D})$ with $\mathsf{v}^c : \mathbb{B}' \to \Phi$ its formulæ assignment, as

$$\mathsf{v}^c(\dot{q}, b, i) = \begin{cases} \mathit{true} & \text{if } i \in \{\mathit{below}, \mathit{notyet}\} \\ \neg\, \mathsf{v}(b) & \text{if } i = \mathit{here}. \end{cases}$$

It follows that if a data tree $\mathbf{t}$ is accepted by $(\mathcal{R}^c, \mathcal{V}^c)$, then any run of $\mathcal{R}$ on $\mathbf{t}$ produces an output such that falsifies $\mathcal{V}$ at some node, and then $\mathbf{t}$ is not accepted by $(\mathcal{R}, \mathcal{V})$. In turn, if $\mathbf{t}$ is accepted by $(\mathcal{R}, \mathcal{V})$, there must be an accepting run of $\mathcal{R}$ whose output verifies $\mathcal{V}$. In other words, there must be a tuple $(\dot{q}, b, \mathit{notyet})$ with $\dot{q}$ an accepting state in the root of every output of $\mathcal{R}$ applied to $\mathbf{t}$, but this cannot be since $\mathcal{R}^c$ does not accept such trees.

Now let us describe with more care how $\mathcal{R}^c$ is built from $\mathcal{R}$. Let $\mathcal{L}_1, \dots, \mathcal{L}_s$ be all the languages from $\mathscr{C}$ used in the transitions of $\mathcal{R}$. For every subset S of these languages consider a language $\mathcal{L}_S$ that tests that a word belongs to all the languages of S, and does not belong to any other language. It follows that $\mathcal{L}_S \in \mathscr{C}$, by closure under finite intersection and complementation.

Let us define the state space of $\mathcal{R}^c$ as $\mathbb{B}'$. Given a language $\mathcal{L}$ over $\dot{Q}$ (the state space of $\mathcal{R}$), we can build a similar language $\mathcal{L}'$ over $\mathbb{B}'$ that tests that we can pick one tuple for each element of the word in such a way that when we project the first component, the word belongs to $\mathcal{L}$. If $\mathcal{L}$ is represented as a regular expression, this boils down to replacing every atomic expression $\dot{q}$ by a big disjunction of all the elements of $\mathbb{B}'$ containing a tuple with state $\dot{q}$. It follows that $\mathcal{L}' \in \mathscr{C}$ since it is closed under inverse homomorphisms. Note that we can further check that there is a word of $\mathbb{B}'^*$ in this language containing a tuple $(\dot{q}, b, i)$ with $i = \mathit{notyet}$ (or $i = \mathit{below}$). This is a consequence of having the membership languages for each tuple with *notyet* and *below*.

Then, for every $A \subseteq \{\mathcal{L}_1, \dots, \mathcal{L}_s, \mathit{notyet}, \mathit{below}\}$ consider a language $\mathcal{L}_A$ that tests

—that there is a word over $\mathbb{B}'$ and a tuple for every element such that the projection on $\dot{Q}$ is accepted by all languages of S and rejected by all others;

—if $\mathit{notyet} \in A$ that there is an element of the word containing a tuple with *notyet*;

—and if $\mathit{below} \in A$ that there is an element of the word containing a tuple with *below*.

Now, $\mathcal{R}^c$ is built with the set of all rules of the form $(B, a, B, \mathcal{L}_A)$ where $A \subseteq \{\mathcal{L}_1, \dots, \mathcal{L}_s, \mathit{here}, \mathit{below}\}$, $a \in \mathbb{A}$, and $B \in \mathbb{B}'$ such that

for every $\dot{q} \in \dot{Q}$, $b \in \mathbb{B}$ and $\mathcal{L} \in A$ such that $(\dot{q}, a, b, \mathcal{L}) \in \delta$ then either
—$below \notin A$ and there is $i \in \{here, notyet\}$ where $(\dot{q}, b, i) \in B$, or
—$below \in A$ and $(\dot{q}, b, below) \in B$.

Thus, *below* means that there was a *here* in the subtree, and *here* means that it is in that precise point that a formula is falsified.

Finally, the accepting states are those whose every tuple with an accepting state contains no *notyet* flag. □

Proof of Proposition 4.10. We first show (1). Note that $\mathbf{t}'$ is the result of adding some subtrees $\mathbf{t}|_{x\cdot i_1}, \dots, \mathbf{t}|_{x\cdot i_\ell}$, where $\ell = \#children(\mathbf{t}', x) - \#children(\mathbf{t}, x)$, plus some possible reordering of the siblinghood. Let $\mathbf{t}_0, \mathbf{t}_1, \dots, \mathbf{t}_\ell$ be data trees such that $\mathbf{t}_0 = \mathbf{t}$, and $\mathbf{t}_j$ with $j > 0$ is the result of adding $\mathbf{t}|_{x\cdot i_j}$ as a last child of x in $\mathbf{t}_{j-1}$. Observe that we can apply Proposition 4.9 to each of the pair of trees $(\mathbf{t}_0, \mathbf{t}_1), (\mathbf{t}_1, \mathbf{t}_2), \dots, (\mathbf{t}_{\ell-1}, \mathbf{t}_\ell)$ obtaining that for every position $y \in \mathsf{pos}(\mathbf{t}_j)$ and $j \in [\ell]$,

$$\mathrm{desc}_{\mathbf{t}_j|_y}(d) = \mathrm{desc}_{\mathbf{t}_{j-1}|_{g^j_x(y)}}(d)$$

where $g^j_x : \mathsf{pos}(\mathbf{t}_j) \to \mathsf{pos}(\mathbf{t}_{j-1})$ is the surjective function given by Proposition 4.9. Then, the function $g_x = g^1_x \circ \cdots \circ g^\ell_x$ is surjective onto $\mathsf{pos}(\mathbf{t})$, and for every position $y \in \mathsf{pos}(\mathbf{t}_\ell)$, $\mathrm{desc}_{\mathbf{t}_\ell|_y}(d) = \mathrm{desc}_{\mathbf{t}|_{g_x(y)}}(d)$. Now, note that $\mathbf{t}_\ell$ and $\mathbf{t}'$ (and equivalently f_x and g_x) differ only in the order of the subtrees of position x. Since, by definition, desc is invariant under reordering of siblings, it follows that (1) holds.

To show (2), first consider any position of the form $x\cdot i \in \mathsf{pos}(\mathbf{t}')$, and observe that $\mathbf{t}'|_{x\cdot i}$ has a run $(\rho \circ f_x)|_{x\cdot h_x(i)}$. This is a consequence of $\mathbf{t}'|_{x\cdot i}$ and $\mathbf{t}|_{x\cdot h_x(i)}$ being identical, and the automaton being bottom-up. Secondly, we prove that $\mathbf{t}'|_x$ has a run $(\rho \circ f_x)|_x$. This is because $\rho(x\cdot h_x(1)) \cdots \rho(x\cdot h_x(n)) \in \mathcal{L}$ by hypothesis and hence $(\rho \circ f_x)|_x(1) \cdots (\rho \circ f_x)|_x(n) \in \mathcal{L}$, where $n = \#children(\mathbf{t}', x)$. We can then apply the transition $(\rho(x), \mathbf{a}(x), \mathbf{b}(x), \mathcal{L})$ obtaining $\rho(x)$ at the root. Finally, since x has the same state in ρ and $\rho \circ f_x$, and the trees $\mathbf{t}$ and $\mathbf{t}'$ are isomorphic except for perhaps the subtree rooted on x, it follows that $\rho \circ f_x$ is a run on $\mathbf{t}'$. □

Proof of Lemma 4.16. Given $(\mathcal{A}, q') \in I'$, $(\mathcal{A}, q) \in I$ as in the Lemma, and $d \in [\![\mathcal{A}, q']\!]^{\mathbf{t}|_{y\cdot i}}$, we show that $d \in [\![\mathcal{A}, q]\!]^{\mathbf{t}|_x}$. Let z be such that $\mathbf{d}(z) = d$ and $q' \xrightarrow[y\cdot i, z]{\mathcal{A}} q_f$ with $q_f \in \mathcal{Q}^{\mathcal{A}}_F$. By hypothesis, $q \xrightarrow[x, y]{\mathcal{A}} q'$. Hence, $q \xrightarrow[x, y]{\mathcal{A}} q' \xrightarrow[y\cdot i, z]{\mathcal{A}} q_f$, and then $d \in [\![\mathcal{A}, q]\!]^{\mathbf{t}|_x}$.

Now suppose $d \in [\![I']\!]^{\mathbf{t}|_{y\cdot i}}$. Since for all $(\mathcal{A}, q) \in I$ there exists $(\mathcal{A}, q') \in I'$ as before, with $d \in [\![\mathcal{A}, q']\!]^{\mathbf{t}|_{y\cdot i}}$, we can apply the same reasoning, obtaining that $d \in [\![I]\!]^{\mathbf{t}|_x}$. Thus, $[\![I']\!]^{\mathbf{t}|_{y\cdot i}} \subseteq [\![I]\!]^{\mathbf{t}|_x}$. □

Proof of Lemma 4.18. We show that $\mathsf{L} := (\mathsf{K}\cdot\mathsf{N})^{\mathsf{R}}\cdot(\mathsf{V}\cdot\mathsf{R}) + \mathsf{K}\cdot\mathsf{N}$ is a good upper-bound. This is a direct consequence of the cardinality of *Inters* and the number of possible pairs $(\mathcal{A}, q)$. To verify the validity condition as in Definition 4.5, for every $I \in Inters$ out of at most $(\mathsf{K} \cdot \mathsf{N})^{\mathsf{R}}$, we might need to add up to $\mathsf{V} \cdot \mathsf{R}$ certificates in the children to witness I: one for each of the V data values and each $(\mathcal{A}, q) \in I$. On the other hand, to verify the inductivity condition on $\kappa(x)$, we need at most one extra child witness for each automaton and state in $\mathrm{desc}_{\mathbf{t}|_x}(\kappa(x))$, which can have no more than $\mathsf{K} \cdot \mathsf{N}$ elements. □

Proof of Corollary 4.20. We show that it can be bounded by

$$\begin{aligned} \mathsf{W} &:= (|\tilde{Q}|-1)\cdot(\mathsf{L}+1) \quad \text{which by definition of } \mathsf{L} \\ &= (|\tilde{Q}|-1)\cdot((\mathsf{K}\cdot\mathsf{N})^{\mathsf{R}}\cdot p'(\mathsf{V},\mathsf{R},\mathsf{K},\mathsf{N})+1) \\ &= (\mathsf{K}\cdot\mathsf{N})^{\mathsf{R}}\cdot p(\mathsf{V},\mathsf{R},\mathsf{K},\mathsf{N},|\tilde{Q}|) \end{aligned} \tag{6}$$

for some polynomials p, p'. Given a maximal sequence of siblings $x{\cdot}1, \dots, x{\cdot}l$, we already established in Lemma 4.18 that a valid and inductive subset $\mathcal{C}_x$ needs no more than L elements to preserve the correctness of the certificate. Hence, there can be at most $\mathsf{L}+1$ zones of consecutive positions not in $\mathcal{C}_x$, and each one of them must have at most $|\tilde{Q}|-1$ elements. Should it have more, then there are at least two repeating elements with the same horizontal configuration by the pigeonhole principle, and we can then apply Lemma 4.19 and remove some of the siblings. □

Proof of Lemma 4.24. The right-to-left implication is a direct consequence of the definition of $\hbar'(I)$. For the left-to-right implication, suppose that $|[\![I]\!]^{\mathbf{t}_0}| \geq k$. Take $d \in [\![I]\!]^{\mathbf{t}_0}$. We argue that if d appears in some $\hat{\kappa}_i(\epsilon)$, then it is in $\hbar'(I)$. This is because all the subtrees $\mathbf{t}_i$ with $d \in \mathit{data}(\mathbf{t}_i)$ must have $d \in C_i$, as a consequence of the hypothesis (2). Hence we have a *complete* description of d at every subtree, and f' yields the correct description of d at the root by Lemma 4.23, which implies that $d \in \hbar'(I)$ by definition of $\hbar$.

Suppose on the other hand that d is not in $\hat{\kappa}_i(\epsilon)$ for any i. Since every κ_i is correct, this means that d does not appear in any small intersection $[\![I']\!]$ of fewer than V data values. Hence, in particular $|[\![I]\!]^{\mathbf{t}_0}| \geq \mathsf{V}$. By hypothesis (2), we know that d can appear in at most one tree $\mathbf{t}_i$, or at the root. We consider both cases.

If d appears only at the root, it means that $d = d_0$, and for every $(\mathcal{A}, q) \in I$ there is $q' \in \mathcal{Q}_F^{\mathcal{A}}$ with $(q, b, q') \in \mathcal{A}$ and this is taken into account by the definition of $\hbar'(I)$. Hence, $d \in \hbar'(I)$.

If d appears only in $\mathbf{t}_i$, this means that there is an intersection I' such that $d \in [\![I']\!]^{\mathbf{t}_i}$ and for every $(\mathcal{A}, q) \in I$ there is $(\mathcal{A}, q') \in I'$ with $(q, b, q') \in \mathcal{A}$. Note that if $|[\![I']\!]^{\mathbf{t}_i}| < \mathsf{V}$ then $\hat{\kappa}_i(\epsilon) \supseteq [\![I']\!]^{\mathbf{t}_i}$. Since we had assumed that $d \notin \hat{\kappa}_i(\epsilon)$, we have that $|[\![I']\!]^{\mathbf{t}_i}| \geq \mathsf{V}$. Then, there are V data values $d_1, \dots, d_{\mathsf{V}} \in \hat{\kappa}_i(\epsilon) \cap [\![I']\!]^{\mathbf{t}_i}$. By Lemma 4.16, $[\![I']\!]^{\mathbf{t}_i} \subseteq [\![I]\!]^{\mathbf{t}_0}$ and thus $d_1, \dots, d_{\mathsf{V}} \in [\![I]\!]^{\mathbf{t}}$. This means that $d_1, \dots, d_{\mathsf{V}}$ are described in M and hence that $d_1, \dots, d_{\mathsf{V}} \in \hbar'(I)$ and $|\hbar'(I)| \geq \mathsf{V}$. Finally, note that d cannot appear in some $\mathbf{t}_i$ *and* at the root by hypothesis (3). □

Proof of Lemma 4.23. The fact that $d \in C_1 \cup \dots \cup C_m$ implies that $d \in \hat{\kappa}_i(\epsilon)$ for some i. Then, by hypothesis (2), if $d \in \mathit{data}(\mathbf{t}_j)$ for some $j \in [m]$, $j \neq i$, then $d \in \hat{\kappa}_j(\epsilon)$. This means that we have a complete description of d at every element of M: for every $j \in [m]$ either $\alpha_j(d) = \mathrm{desc}_{\mathbf{t}_j}(d)$, or $\alpha_j(d) = \bot$ and $\mathrm{desc}_{\mathbf{t}_j}(d) = \emptyset$. Hence, by definition of f' we have: $f'(d) = \mathrm{desc}_{\mathbf{t}_0}(d)$. □

Proof of Lemma 4.28. Now we have at most $|\mathbb{D}'|$ data values, and the function of each configuration is restricted to the data values (at most W) of the configuration. We then have the following bounds by definition of $\mathit{TConfigs}'$ (5).

$$|\mathit{TConfigs}'| \leq |\dot{Q}| \cdot |\mathbb{D}'| \cdot 2^{\mathsf{K}\cdot\mathsf{N}} \cdot (|\mathbb{D}'| \cdot 2^{\mathsf{K}\cdot\mathsf{N}})^{\mathsf{W}}$$

$|\mathbb{D}'|$ is polynomial in W, and we then have

$$|\mathit{TConfigs}'| \leq |\dot{Q}| \cdot (p(\mathsf{W}))^{q(\mathsf{W})} \cdot 2^{\mathsf{K} \cdot \mathsf{N} \cdot r(\mathsf{W})} \tag{7}$$

for some polynomials p, q, r. W is exponential only in R, and if R is constant, W is polynomial in $|\mathcal{R}|$ and $|\mathcal{V}|$ by definition (6). Thus, the lemma follows. $\square$

Proof of Lemma 4.29. Condition (i) can be tested in polynomial time in $|\mathcal{R}|$, $|\tilde{Q}|$ and n, where $|\tilde{Q}|$ is polynomial in W. The function $f(T_1 \cdots T_n)(b, d_0)(d)$ can be computed in polynomial time in Aut and $|T_1|+\cdots+|T_n|$ (which is polynomial in W). It follows that the description function α of condition (vi) can be checked in polynomial time in W, since it consists in at most $\mathsf{W}+1$ applications of $f(M)(b, d_0)(d)$. Conditions (ii) and (iii) are easily checked in polynomial time in $|T|, |T_1|, \ldots, |T_n|$. Condition (v) can be tested in polynomial time in W and Aut by the above reasons. For condition (iv), we see that $\hbar(M)(b, d_0)(I)$ can be built in time polynomial, and that for every possible intersection I and value of $\hbar(M)(b, d_0)(I)$ a polynomial condition must be checked, hence, since $|\mathit{Inters}|$ is polynomial in W, testing condition (iv) remains polynomial. $\square$

Proof of Corollary 6.3. The fact that $\mathsf{R} = 2$ and $\mathsf{V} = 2$ is immediate from the definition of $\mathcal{V}$. As already discussed, the transducer only uses one state, $|\dot{Q}| = 1$, and that it only needs to use transitions $(\dot{q}, a, b, \mathcal{L})$ with $\mathcal{L} = (\dot{Q})^*$. $\mathcal{L}$ is represented with a NFA with a set of states $|\tilde{Q}| = 1$.

The translation we presented is not polynomial. Indeed there is an exponential number of transitions both in $\mathcal{R}$ and in the automata $\mathcal{A}_\alpha$ of $\mathcal{V}$, but it contains a *polynomial* number of automata $\mathcal{A}_\alpha$. Further, each $\mathcal{A}_\alpha$ has a number of states polynomial in $|\alpha|$, and a number of transitions polynomial in $|\alpha|$ and $|\mathbb{B}|$. This last one is singly exponential in $|\varphi|$. $\square$

Proof of Proposition 6.7. Suppose that if a formula $\eta \in \mathscr{P}$ is satisfied by a data tree, then it is satisfied by a tree of height $h \leq p(|\eta|)$, where p is a fixed polynomial that does not depend on η.

We make use of the translation of Theorem 6.2, but we avoid storingthe transition relations of $\mathcal{R}$ and of the automata Aut of $\mathcal{V}$ explicitly. Instead, we use two facts. First, that testing whether a set S of subformulæ of η is a locally consistent set uses polynomial space in $|\eta|$, and these sets can then be enumerated in polynomial space. And second, that $\mathsf{v}(b)$ can be built in polynomial space, given a locally consistent set b. Thus, the space $\mathit{sp}(\vdash)$ needed for checking $T_1 \cdots T_n \vdash T$ is polynomial.

Hence, by Theorem 4.33 with Remark 4.34 we have an emptiness algorithm that uses space $p(h, \mathsf{K}, \mathsf{N}) + \mathit{sp}(\vdash)$ for some polynomial p. This is a consequence of $|\dot{Q}|$, V, R and $|\tilde{Q}|$ being constant, and $h, \mathsf{K}, \mathsf{N}$ being polynomial in $|\eta|$ by Corollary 6.3. $\square$

Proof of Proposition 6.8. Benedikt et al. [2008] show that $\mathsf{XPath}(\downarrow, =)$ is PSpace-hard and NExpTime-easy. Here we show a PSpace upper bound by proving the poly-depth model property. Note that if η is satisfiable in $\mathbf{t}$, then it is satisfiable in $\mathbf{t} \upharpoonright n$ where n is the maximum number of nested $\downarrow$ in η[9], and $\mathbf{t} \upharpoonright n$ is the submodel of $\mathbf{t}$ consisting of all the nodes that are at distance at most n from

[9]For example, the maximum number of nexted $\downarrow$ in the expression $\langle \downarrow [a \wedge \langle \downarrow\downarrow [b] \rangle] \downarrow [\langle \downarrow\downarrow [a] \rangle] \rangle$ is 4.

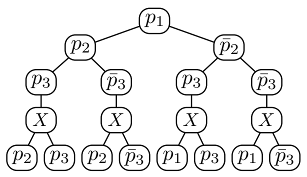

Fig. 11. A coding of the satisfiability of a formula $\exists p_1 \forall p_2, p_3.(p_1 \vee p_2) \wedge (p_3 \vee \neg p_3 \vee \neg p_2)$.

the root: $\mathbf{t} \upharpoonright n = \{x \mapsto \mathbf{t}(x) \mid x \in \mathsf{pos}(\mathbf{t}), |x| \leq n\}$. Hence, by Proposition 6.7, $\mathsf{XPath}(\downarrow, =)$ is in PSPACE. $\square$

PROOF OF PROPOSITION 6.10. The proof goes by reduction from an instance of the QBF (Quantified Boolean Formula [Garey and Johnson 1979]) validity problem to SAT-$\mathsf{XPath}(\downarrow_*)$.

Let

$$\varphi = Q_1 p_1 \ldots Q_n p_n.\psi$$

where p_i are propositional variables (pairwise distinct), $Q_i \in \{\forall, \exists\}$ and ψ is a formula of the propositional calculus in CNF.

The idea is to force a model in which every branch contains a full valuation for the variables $p_1, \ldots, p_n$. The tree's alphabet is $\mathbb{A} = \{p_1, \ldots, p_n, \bar{p}_1, \ldots, \bar{p}_n, X\}$, and every branch lists a valuation in order, that is, first there is a node with a label in $\{p_1, \bar{p}_1\}$, then another in $\{p_2, \bar{p}_2\}$, etc. The label X simply marks the ending of a valuation in a branch. After this marking we build the tree that satisfies ψ by choosing a witness atom for every disjunctive clause. Finally, we check that there are no inconsistencies with respect to its valuation. For example, in Figure 11 we depict a possible tree that is forced by a formula.

Let v_i be the formula that specifies that the node is a valuation for the propositional variable p_i, $v_i := p_i \vee \bar{p}_i$. We specify formulæ $f_1, \ldots, f_n$ depending on $Q_1, \ldots, Q_n$, where G is defined as in the proof of Theorem 6.5.

—If $Q_1 = \forall$, then $f_1 = \langle \downarrow_*[p_1] \rangle \wedge \langle \downarrow_*[\bar{p}_1] \rangle$.
If $Q_1 = \exists$, then $f_1 = \langle \downarrow_*[p_1] \rangle \vee \langle \downarrow_*[\bar{p}_1] \rangle$.

—If $Q_i = \forall$ for $i > 1$, then $f_i = \mathsf{G}(v_{i-1} \to \langle \downarrow_*[p_i] \rangle \wedge \langle \downarrow_*[\bar{p}_i] \rangle)$.
If $Q_i = \exists$, then $f_i = \mathsf{G}(v_{i-1} \to \langle \downarrow_*[p_i] \rangle \vee \langle \downarrow_*[\bar{p}_i] \rangle)$.

—φ_X forces that the label X always appears once the valuation for all propositions has been defined.

$$\varphi_X = \neg \langle \downarrow_*[v_1] \downarrow_*[v_2] \cdots \downarrow_*[v_{n-1}] \downarrow_*[v_n \wedge \neg \langle \downarrow_*[X] \rangle] \rangle$$

—For all X we build a formula for $\psi = C_1 \wedge \ldots \wedge C_l$ where $C_i = t_1 \vee \ldots \vee t_m$ and each t_j is either p_k or $\bar{p}_k$ for some k. That is,

$$\tau = \bigwedge_{C_i} \bigvee_{t \in C_i} \langle \downarrow_*[t] \rangle$$

and this must hold for all X-valued nodes,

$$\varphi_\psi = \mathsf{G}(X \to \tau) \ .$$

—Finally, we must check that there are no inconsistencies between the p_i along a branch.

$$\varphi_{inc} = \bigwedge_{i=1}^{n} \neg\langle\downarrow_*[p_i]\downarrow_*[\bar{p}_i]\rangle \wedge \neg\langle\downarrow_*[\bar{p}_i]\downarrow_*[p_i]\rangle$$

The final formula is then

$$\varphi_F = \bigwedge_{i=1}^{n} f_i \wedge \varphi_X \wedge \varphi_\psi \wedge \varphi_{inc} \ .$$

Suppose that φ is valid. We can build a tree that satisfies φ_F as follows. Let's assume that $Q_1 = \exists$ (the other case is similar). Note that $\exists p_i Q_{i+1} p_{i+1} \cdots Q_n p_n.\psi$ is valid if, and only if, there is a valuation $v(p_i)$ for p_i such that $Q_{i+1}p_{i+1}\cdots Q_n p_n.\psi[p_i \mapsto v(p_i)]$ is valid, (where this is interpreted as the result of replacing p_i by true or false in the formula according to $v(p_i)$). Conversely, $\forall p_i Q_{i+1}p_{i+1}\cdots Q_n p_n.\psi$ is valid if, and only if, both $Q_{i+1}p_{i+1}\cdots Q_n p_n.\psi[p_i \mapsto 1]$ and $Q_{i+1}p_{i+1}\cdots Q_n p_n.\psi[p_i \mapsto 0]$ are valid. Now, we build the tree as follows. We first take the witnessing valuation of p_1 for the validity of φ and put it as the root. Then we iterate adding leaves in the tree until obtaining a tree whose every leaf is at depth $n-1$ in the following way. Suppose we are adding the children of a leaf at depth i with $i < n-1$. Let v be the valuation of $p_1 \ldots p_i$ that corresponds to the labels that appear in the path that leads to the current leaf. Consider φ' as the result of replacing every p_j by true or false corresponding to $v(p_j)$ for every $j \leq i$ in $Q_{i+1}p_{i+1}\cdots Q_n p_n\psi$, and note that φ' must be valid. If $Q_{i+1} = \exists$ we just add one child with the witnessing valuation $v(p_{i+1})$ that makes $\varphi'[p_{i+1} \mapsto v(p_{i+1})]$ valid. If $Q_{i+1} = \forall$ we add *two* children with labels p_{i+1} and $\bar{p}_{i+1}$. Once we do this with all the quantifiers, we append a node with label X to all leaves, and put the witness of the satisfaction of ψ which must be a choice between the valuations that occurred along the path, which gives us a tree whose every leaf is at depth $n+1$, as in Figure 11. Note that it cannot be that there is a p_i and $\bar{p}_i$ along a path. Indeed, this tree satisfies φ_F.

On the other hand, if φ_F is satisfied by a tree, then we can produce witness for every $\exists$ quantifier of the formula, always making sure that the valuation chosen for one propositional variable p_i will not change, since we disallow having p_i nad $\bar{p}_i$ in the same path. This shows that φ is in fact valid.

Then we have that φ is QBF-valid if and only if φ_F is satisfiable. Note that this reduction does not use path unions. Then, this lower bound holds even in the absence of path unions. $\square$

Proof of Lemma 6.11. All path formulæ that hold at the root of $\mathbf{t}$, hold also in $\mathbf{t}'$ as it is an *extension* of the tree. On the other hand, any path formula that is satisfied at the root by a succession of nodes in a branch in $\mathbf{t}'$, can also be found in $\mathbf{t}'$, as we only count with the $\downarrow_*$ axis. This is true not only for the root but for any position of $\mathbf{t}$. In other words, the logic $\mathsf{XPath}(\downarrow_*)$ is closed under *subtree copy*. $\square$

Proof of Proposition 6.12. We prove that $\varphi \in \mathsf{XPath}(\downarrow_*)$ is satisfiable iff it is satisfied by a tree of height bounded by $|\varphi|^2$.

For the proof of this statement we first define, for a path expression, the set of possible sequences of node tests that it must satisfy, that we note with '*nseq*' (Table V). The idea is that if for instance $\{\psi,\varphi\}\cdot\{\varphi\}\cdot\{\psi,\eta\} \in nseq(\alpha)$ for a path

$$\begin{array}{rlr}
nseq : \mathsf{XPath}(\downarrow_*) \to \wp\big((\wp(\mathsf{XPath}(\downarrow_*)))^*\big) & & \\
nseq(\alpha \cup \beta) = nseq(\alpha) \cup nseq(\beta) & & nseq([\psi]) = \{\{\psi\}\} \\
nseq(\alpha\beta) = \{S_1{\cdot}(A_1 \cup A_2){\cdot}S_2 \mid & & nseq(\varepsilon) = \{\emptyset\} \\
S_1{\cdot}A_1 \in nseq(\alpha), A_2{\cdot}S_2 \in nseq(\beta)\} & & nseq(\downarrow_*) = \{\emptyset{\cdot}\emptyset\}
\end{array}$$

Table V. Given a path expression α, $nseq(\alpha)$ is the set of possible sequence of node tests that a witnessing branch must satisfy.

expression α, then $\mathbf{t} \models \langle\alpha\rangle$ if there are $\epsilon \preceq x \preceq y$ such that $\mathbf{t} \models \psi$, $\mathbf{t} \models \varphi$, $\mathbf{t}|_x \models \varphi$, $\mathbf{t}|_y \models \psi$ and $\mathbf{t}|_y \models \eta$. Let $\mathsf{wit}_\mathbf{t} : \mathsf{pos}(\mathbf{t}) \times \mathsf{XPath}(\downarrow_*) \to \wp(\mathsf{pos}(\mathbf{t}))$ be a *witness* function such that for any $x \in \mathsf{pos}(\mathbf{t})$ and $\alpha \in \mathsf{XPath}(\downarrow_*)$ such that $\mathbf{t}|_x \models \langle\alpha\rangle$ there is $S \in nseq(\alpha)$ where

—all elements in $\mathsf{wit}_\mathbf{t}(x,\alpha) = \{x_1,\dots,x_n\}$ belong to the same branch, $x_1 \prec x_2 \prec \cdots \prec x_n$;

—there are $i_1 \le \cdots \le i_{|S|}$ such that $\{i_1,\dots,i_{|S|}\} = \{1,\dots,n\}$, and $\mathbf{t}|_{x_{i_j}} \models \bigwedge S(j)$ for every j.

Note that in particular $|\mathsf{wit}_\mathbf{t}(x,\alpha)| \le |S|$ for some $S \in nseq(\alpha)$, and we hence have

$$|\mathsf{wit}_\mathbf{t}(x,\alpha)| \le |\alpha| \ . \tag{8}$$

In the sequel, given $\varphi \in \mathsf{XPath}(\downarrow_*)$ we write *nesting*(φ) for the maximum number of nested node tests (that is, of nested '[ ]') that are in the formula φ.

We can make use of Lemma 6.11 to make sure that we can always assume $\mathsf{wit}_\mathbf{t}$ to be in a normal form where all its elements are chained with the parent/child relation. That is, that given $\mathsf{wit}_\mathbf{t}(x,\alpha) = \{x_1 \prec \cdots \prec x_n\}$, then for all i

$$x_{i+1} = x_i{\cdot}j \text{ for some } j \ . \tag{9}$$

We are now in a position to explain the main argument. Let $\varphi \in \mathsf{XPath}(\downarrow_*)$ and $n = nesting(\varphi)$, and suppose we have a tree $\mathbf{t}$ that satisfies φ. Next, we describe a procedure to 'mark' the important nodes in the tree. We start by marking the root with the label 'n'. Then, for every position x marked with $t \ge 0$ and for every path expression $\beta \in \mathsf{sub}(\varphi)$ such that $nesting(\beta) \le t$ and $\mathbf{t}|_x \models \beta$, we mark all positions $y \in \mathsf{wit}_\mathbf{t}(x,\beta)$ with '$t-1$'.

Note that all the positions marked with $\{-1,0,\dots,n\}$ form *one* connected component by (9), and that they are all at a distance from the root of at most $nesting(\varphi)\cdot|\varphi|$ by (8). Let $\mathbf{t}'$ be the tree resulting from eliminating all the positions with no marking. We then have that $\mathbf{t} \models \varphi$ iff $\mathbf{t}' \models \varphi$, and hence that $\mathsf{XPath}(\downarrow_*)$ has the poly-depth model property. We conclude by Proposition 6.7 that $\mathsf{XPath}(\downarrow_*)$ is in PSPACE. $\square$

PROOF OF PROPOSITION 6.15. The proof of the upper-bound goes by showing the poly-depth model property. Let $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$ be a data tree. The key observation is that any $\mathsf{XPath}^{\not\approx}(\downarrow_*,=)$ expression of the form $\langle\alpha\rangle$, $\langle\alpha = \beta\rangle$ or $\langle\alpha \neq \beta\rangle$ that is satisfied at a node x of a tree, is also satisfied in *any ancestor* of x. This is because all path expressions start with a $\downarrow_*$ axis. In other words, for any pair of positions x, x' such that $x \preceq x'$, the set of formulæ of the aforementioned type that are satisfied in x' is a *subset* of those that are satisfied in x. Given a branch $\epsilon = x_0 \prec \cdots \prec x_n$ of $\mathbf{t}$ where x_{i+1} is a child of x_i for all i, and given a formula φ, consider for every $i \in [n]$

$$C_i = \{\psi \mid \psi \in \mathsf{sub}(\varphi),\ \psi \text{ of the form } \langle\alpha\rangle,\ \langle\alpha = \beta\rangle \text{ or } \langle\alpha \neq \beta\rangle \text{ s.t. } \mathbf{t}|_{x_i} \models \psi\}.$$

We then have $C_0 \supseteq \cdots \supseteq C_n$, and there is only a *polynomial* number of different sets. Take any two $C_i = C_j$ such that $\mathbf{a}(x_i) = \mathbf{a}(x_j)$. Then, the tree which results from replacing the subtree at x_i by the subtree at x_j preserves the satisfaction of φ at the root. Hence, the logic has the poly-depth model property and by Proposition 6.7 its satisfiability problem is in PSpace.

The lower-bound comes from the proof of Proposition 6.10, whose encoding is in $\mathsf{XPath}^{\not\epsilon}(\downarrow_*, =)$. □

Proof of Theorem 4.33. Given the previous algoritm of Theorem 4.30, note that all the configurations corresponding to derivations of height i are present once the ith iteration has been executed. Also, note that the height of the derivation and the height of the tree are related in this sense: any tree of height h can be witnessed by a derivation of height h. We can then build a top-down NPSpace algorithm as follows.

Let us denote by $g(T, l)$ the following procedure for $T \in \mathit{TConfigs}'$, $l \in [0..h]$. First, $g(T, 0)$ yields 'ok' iff $\epsilon \vdash T$, otherwise it fails. $g(T, l + 1)$ performs the following tasks. Guesses the tree configurations $T_1, \ldots, T_n$ with $0 \leq n \leq \mathsf{W}$. Checks $T_1 \cdots T_n \vdash T$ in PSpace, and recursively tests that $g(T_1, l), \ldots, g(T_n, l)$ succeed. Consider now the main algorithm that guesses a root configuration $T \in \mathit{TConfigs}'$ and checks both that it contains a final state $\dot{q} \in \dot{Q}_F$ and that $g(T, h)$ succeeds. This algorithm correctly answers whether there exists a derivation of height at most h that has T at the root.

Notice that the space needed to store the data description function α of a configuration T is bounded by $sp(\alpha) = (\mathsf{W} + 1) \cdot \log(|\mathbb{D}'|) \cdot \mathsf{K} \cdot \mathsf{N}$, for $\mathbb{D}'$ as defined in (4). Then the space needed to store a tree configuration is

$$\begin{aligned} sp(T) &= \log(|\dot{Q}|) + \log(|\mathbb{D}'|) + sp(\alpha) \\ &\leq \log(|\dot{Q}|) + p(|\tilde{Q}|, \mathsf{K}, \mathsf{N}, \mathsf{V}) \end{aligned}$$

for some polynomial p. This is a consequence of (4) and (6), and the fact that R is fixed. If we perform a DFS evaluation strategy of g we only need to store simultaneously at most $(\mathsf{W} + 1) \cdot h$ configurations and hence the algorithm takes a space polynomial in the size of the automata $\mathcal{R}$, $\mathcal{V}$ considering R is fixed. It is then immediate that this is a NPSpace procedure for any configuration and l.

Thus, as NPSpace = PSpace the theorem follows. □